

Aggregations over Generalized Hypertree Decompositions

Manas Joglekar¹, Rohan Puttagunta¹, and Chris Ré¹

¹ Department of Computer Science

Stanford University

{manasrj, rohanp, chrismre} @ cs.stanford.edu

Abstract

We study a class of aggregate-join queries with multiple aggregation operators evaluated over annotated relations. We show that straightforward extensions of standard multiway join algorithms and generalized hypertree decompositions (GHDs) provide best-known runtime guarantees. In contrast, prior work uses bespoke algorithms and data structures and does not match these guarantees. Our extensions to the standard techniques are a pair of simple tests that (1) determine if two orderings of aggregation operators are equivalent and (2) determine if a GHD is compatible with a given ordering. These tests provide a means to find an optimal GHD that, when provided to standard join algorithms, will correctly answer a given aggregate-join query. The second class of our contributions is a pair of complete characterizations of (1) the set of orderings equivalent to a given ordering and (2) the set of GHDs compatible with some equivalent ordering. We show by example that previous approaches are incomplete. The key technical consequence of our characterizations is a decomposition of a compatible GHD into a set of (smaller) *unconstrained* GHDs, i.e. into a set of GHDs of sub-queries without aggregations. Since this decomposition is comprised of unconstrained GHDs, we are able to connect to the wide literature on GHDs for join query processing, thereby obtaining improved runtime bounds, MapReduce variants, and an efficient method to find approximately optimal GHDs.

1 Introduction

Generalized hypertree decompositions (GHDs), introduced by Gottlob et al. [9, 10] and further developed by Grohe and Marx [12], provide a means for performing early projection in join processing, which can result in dramatically faster runtimes. In this work, we extend GHDs to handle queries that include aggregations, which allows us to capture both SQL-aggregate processing and message passing problems. Motivated by our own database engine based on GHDs [1, 22, 26], we seek to more deeply understand the space of optimization for aggregate-join queries.

We build upon work by Green, Karvounarakis, and Tannen [11] on annotated relations to define our notion of aggregation. These annotations provide a general definition of aggregation, allowing us to represent a wide-ranging set of problems as aggregate-join queries. Our queries, which we call AJAR (Aggregations and Joins over Annotated Relations) queries, contain semiring quantifiers that “sum over” or “marginalize out” values. We formally define AJAR queries in Section 3, but they are easy to illustrate by example:

► **Example 1.** Consider two relations with attributes $\{A, B\}$ and $\{B, C\}$ such that each tuple is annotated with some integer; we call these relations \mathbb{Z} -relations. Consider the query:

$$\sum_C \sum_B R(A, B) \bowtie S(B, C)$$

Our output will then be a \mathbb{Z} -relation with attribute set $\{A\}$. Each value a of attribute A in R is associated with a set X_a of pairs (b, z_R) composed of a value b of attribute B and an annotation



licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

| R | | | S | | | $R \bowtie S$ | | | $\sum_C \sum_B R \bowtie S$ | | |
|-----|---|---|-----|---|---|---------------|---|---|-----------------------------|---|----|
| A | B | Z | B | C | Z | A | B | C | Z | A | Z |
| 1 | 3 | 3 | 1 | 1 | 4 | | | | 18 | | |
| 1 | 2 | 1 | 3 | 3 | 6 | | | | 8 | | |
| 1 | 1 | 2 | | | | | | | | 1 | 26 |

■ **Figure 1** Illustrating the computation of Example 1

z_R . Furthermore for each b value in X_a , there is a set X_b from relation S of pairs (c, z_S) composed of a value c of attribute C and an annotation z_S . Given X_a and each X_b associated with a given value a , the annotation associated with a in our output will simply be

$$\sum_{b, z_1 \in X_a} \sum_{c, z_2 \in X_b} z_1 * z_2.$$

AJAR queries capture both classical SQL-style queries and newer data processing problems like probabilistic inference via message passing on graphical models [14]. In fact, Aji and McEliece proposed the “Marginalize a Product Function” (MPF) problem [4], which is a special case of an AJAR query, and showed how the problem and its solution capture a number of classic problems and algorithms, including fast Hadamard transforms, Viterbi’s algorithm, forward-backward algorithm, FFT, and probabilistic inference in Bayesian networks. These algorithm are fundamental to various fields; for example the forward-backward algorithm over conditional random fields forms the basis for state of the art solutions to named entity recognition, part of speech tagging, noun phrase segmentation, and other problems in NLP [25]. We are motivated by the wide applicability of queries over annotated relations; annotated relations may provide a framework for combining classical query processing, linear algebra, and statistical inference in a single data processing system.

We consider a generalization of MPF with multiple aggregation operators. We represent an aggregate-join query as a join Q and an aggregation ordering, which specifies both the ordering and the aggregation of each attribute. Our language directly follows from the work of Abo Khamis, Ngo, and Rudra [15], who investigated the “Functional Aggregate Query” (FAQ) problem. In addition to MPF, FAQ is a generalization of Chen and Dalmau’s QCQ problem [7], in which the only aggregates are logical quantifiers (AND and OR).

The key technical challenge in both problems is characterizing the permissible aggregations orders to answer the query. Chen and Dalmau give a complete characterization of which variable orders are permissible for QCQ via a procedure. We first give a simple (complete) procedure for our more general class of queries with multiple aggregations, and then we provide a complete characterization of permissible orders.

- *A Simple Test for Equivalence:* A query can be thought of as a body Q and a string of attribute-operator pairs α . Given a query Q and two orders α and β , we provide a simple test to determine whether α and β are equivalent (i.e., return the same output for any input database). The technical challenge is that different aggregation operators (e.g., \sum and max) cannot freely commute. We show that attribute-operator pairs can commute for only two reasons: (1) their operators commute or (2) their attributes are “independent” in the query, e.g., in the query $\min_B \max_A \sum_C R(A, B), S(B, C)$ the aggregations involving A and C can commute – even though max and \sum do not commute as operators, the query body renders them *independent* given B . We show that these two conditions are *complete*, which leads to a simple test for equivalence (Algorithm 2).
- *A Simple Test for GHD and Order Compatibility:* We say a GHD is compatible with an ordering if we can run standard join algorithms on the GHD while performing aggregations in

the order given by the ordering. We show that testing for compatibility amounts to verifying that for any two attributes A, B , if the topmost GHD node containing A occurs above the topmost node containing B , then A occurs before B in the ordering.

This pair of results gives us a simple algorithm that achieves the best known runtime results. Given a query (Q, α) , enumerate each order β and each GHD G , checking if α is equivalent to β and G is compatible with β . If so, record the cost of solving the query using G , according to (say) fractional hypertreewidth. Solve the query using the lowest cost (G, β) with a standard join algorithm [12].¹

The preceding simple algorithm runs in time exponential in the query size. But finding the optimal GHD even without aggregation is an NP-hard problem, so the brute force optimizer has essentially optimal runtime. It is easy to implement, and a variant is in our prototype database [1, 26].

The more interesting problem is to characterize the notions of equivalence, mirroring Chen and Dalmau. To that end, we give two new, complete characterizations:

- *A Complete Characterization of Equivalent Orders:* Given an order α and two attribute-operator pairs $x, y \in \alpha$, we describe a set of constraints of the form “*in any order, x must appear after y .*” Our constraints are sound and complete, i.e., a string β satisfies these constraints if and only if it is equivalent to α . In contrast, previous approaches have an incomplete characterization, as shown in Example 59 in the Appendix.
- *A Complete Characterization of GHDs compatible with any Equivalent Order.* Given an order α and a query hypergraph Q , we call a GHD ‘valid’ if it is compatible with any ordering equivalent to α . We give a succinct characterization for all valid GHDs. We then describe a decomposition of the query (Q, α) into a series of *characteristic hypergraphs* (without attached aggregation orderings). GHDs for these hypergraphs can be combined into a valid GHD for the original query. We show that for any “node-monotone”² width function, there is a GHD with optimal width w that can be constructed with this decomposition.³ Treewidth, Fractional hypertreewidth, and Submodular width are all node-monotone.

Conceptually, we think the latter result is especially important for tying our work to existing GHD literature; the result reduces our problem to operating on standard GHDs. Pragmatically, we can apply existing GHD results to our characteristic hypergraphs and obtain the following results for free:

- Based on Grohe and Marx [5], we are able to describe our runtime in terms of classical metrics like fractional hypertreewidth. In turn, we can use standard notions to upper bound the runtime like fractional hypertree width, Marx’s submodular width [17], or Joglekar’s efficiently computable variant [13].
- Based on Afrati et al. [3], who bound the communication costs of join processing in terms of a “width” parameter for GHDs, we can develop efficient MapReduce algorithms for solving AJAR queries.

¹ Two technical notes: (1) methods like submodular width [17] or Joglekar and Ré [13] require that we first partition the instances and then run the above algorithm; (2) FAQ [15] is not output sensitive (it does not use GHDs), and so it handles output attributes less efficiently than the above algorithm, as seen in Example 58.

² Informally, a map is *node monotone* if adding more *nodes* to a graph does not reduce the measure, but additional edges may reduce the measure, see Definition 28.

³ In contrast, FAQ’s decomposition strategy may miss the optimal GHD. Appendix Example 33 shows a case in which using the FAQ decomposition gives a width $2n$ while AJAR obtains width n for $n \geq 1$. We also exhibit a family of queries and instances on which FAQ runs in time $\Omega(N^{3n/2})$ while AJAR runs in time $O(N^n)$ for $n \geq 1$, see Appendix Example 60.

- Based on Marx’s approximation [16] for GHDs, we can find approximately optimal GHDs for the popular fractional hypertreewidth measure in polynomial time.

We get the above results essentially for free from forging this connection to GHDs. We view this simple link as a strength of our approach.

Finally, we discuss an extension to handle “product aggregations” that allows us to aggregate away an attribute *before* we join the relations containing the attribute when the aggregation operator is the multiplication operator of the semiring. FAQ was the first to observe that this special case can improve certain types of logical queries. This opens up a new space of equivalent orderings and valid GHDs; mirroring the above results, we give a simple test and a complete characterization of the valid GHDs for queries that include this aggregation. As a result, we obtain similar improvements in runtime relative to previous work.

Outline. We discuss related work in Section 2. In Section 3, we introduce notation and algorithms that are relevant to our work before defining the AJAR problem and discussing its solution, which involves running existing algorithms on a restricted class of GHDs. Section 4 provides a succinct characterization of all orderings that are equivalent to a given ordering. Section 5 discusses how to connect our work with recent research on GHDs, explaining how to construct valid optimal query plans and how to further improve and parallelize our results. In Section 6, we discuss how to incorporate product aggregations.

2 Related Work

Join Algorithms. The Yannakakis algorithm, introduced in 1981, guarantees a runtime of $O(\text{IN} + \text{OUT})$ for α -acyclic join queries [28]. Modern multiway algorithms can process any join query and have worst-case optimal runtime. In particular, Atserias, Grohe, and Marx [5] derived a tight bound on the worst-case size of a join query given the input size and structure. Ngo et al. [19] presented the first algorithm to achieve this runtime bound, i.e. the first worst-case optimal algorithm. Soon after, Veldhuizen presented Leapfrog Triejoin, a very simple worst-case optimal algorithm that had been implemented in LogicBlox’s commercial database system [27]. Ngo et al. [20] later presented the simplified and unified algorithm GenericJoin (GJ) that captured both of the previous worst-case optimal algorithms.

GHDs. First introduced by Gottlob, Leone, and Scarcello [10], hypertree decompositions and the associated hypertree width generalize the concept of tree decompositions [24]. Conceptually, the decompositions capture a hypergraph’s cyclicity, allowing them to facilitate the selective use of GJ and Yannakakis in the standard hybrid join algorithm GHDJoin. There are deep connections between variable orderings and GHDs [15], which we leverage extensively. Grohe and Marx [12] introduced the idea of fractional hypertree width over GHDs, which bounds the runtime of GHDJoin by $\tilde{O}(\text{IN}^w + \text{OUT})$ (\tilde{O} hides poly-logarithmic factors) for w defined to be the minimum fractional hypertree width among all GHDs.

Semirings and Aggregations. Green, Karvounarakis, and Tannen developed the idea of annotations over a semiring [11]. Our notation for the annotations is superficially different from theirs, solely for notational convenience. We delve into more detail in Section 3. This also has been used as a mechanism to capture aggregation in probabilistic databases [23].

MPF. Aji and McEliece [4] defined the “Marginalize a Product Function” (MPF) problem, which is equivalent to the space of AJAR queries with only one aggregation operator. They showed that MPF generalizes a wide variety of important algorithms and problems, which also implies that AJAR queries are remarkably general. They also provided a message passing algorithm to solve MPF, which has since been refined [14]. We provide runtime guarantees that improve the current state of the art.

Aggregate-Join Queries. There is a standard modification to Yannakakis to handle aggregations [28], but the classic analysis provides only a $O(\text{IN} \cdot \text{OUT})$ bound. Bakibayev, Ko-

cisky, Olteanu, and Zavodny study aggregation-join queries in factorized databases [6], and later Olteanu and Zavodny connected factorized databases and GHDs/GHDJoin [21]. They develop the intuition that if output attributes are above non-output attributes, the $+\text{OUT}$ runtime is preserved; we use the same intuition to develop and analyze AggroGHDJoin, a variant to GHDJoin for aggregate-join queries.

Abo Khamis, Ngo, and Rudra present the “Functional Aggregate Query” (FAQ) problem [15], which is equivalent to AJAR. The FAQ/AJAR problems arose out of discussions between Ngo, Rudra, and Ré at PODS12 about how to extend the worst-case result to queries using aggregation and message passing via Green et al.’s semiring formulation. We originally worked jointly on the problem, but we developed substantially different approaches. As a result, we split our work. We argue the the AJAR approach is simpler, as it yields the best known runtime results in only a few simple statements in Section 3. We also describe new complete characterizations as described above. Pragmatically, these completeness results allow us to connect to more easily to existing literature. We have already implemented the algorithm described here in the related database engine EmptyHeaded [1].⁴ This engine has run motif finding, pagerank, and single-source shortest path queries dramatically faster than previous high-level approaches that take datalog-like queries as input.

A primary application of multiple aggregation operators is quantified conjunctive queries (QCQ) and the counting variant, which can be expressed as AJAR queries over the semiring (\vee, \wedge) with aggregations involving both operators. Here, we follow FAQ’s idea to formulate this as a query with product aggregation. Chen and Dalmau [7] completely characterized the space of tractable QCQ by defining a notion of width that relies on variable orderings. Chen and Dalmau’s width definition includes a complete characterization of the permissible variable orderings for a QCQ instance. Their characterization is similar in spirit to the partial ordering we define in Section 4 that characterizes the space of valid GHDs for an AJAR query. However, their results are focused on tractability rather than the optimal runtime exponents; our characterization extends theirs and has improved runtime bounds.

3 AJAR and A Simple Solution

We start by describing some background material needed to define the AJAR problem. After that, we formally define the AJAR problem and our solution to it.

3.1 Background

We use the classic hypergraph representation for database schema and queries [2]. A hypergraph \mathcal{H} is a pair $(\mathcal{V}, \mathcal{E})$, where \mathcal{V} is a non-empty set of *vertices* and $\mathcal{E} \subseteq 2^{\mathcal{V}}$ is a set of *hyperedges*. Each $A \in \mathcal{V}$ is called an *attribute*. Each attribute has a corresponding *domain* \mathcal{D}^A .

- **Data** For each hyperedge $F \in \mathcal{E}$, there is a corresponding relation $R_F \subseteq \prod_{A \in F} \mathcal{D}^A$; we use the notation \mathcal{D}^F to denote the domain of the tuples $\prod_{A \in F} \mathcal{D}^A$.
- **Join Query** Given a set \mathcal{E} and a relation R_F for each $F \in \mathcal{E}$, let $\mathcal{V} = \cup_{F \in \mathcal{E}} F$. The join query is written $\bowtie_{F \in \mathcal{E}} R_F$ and is defined as

$$\{t \in \mathcal{D}^{\mathcal{V}} \mid \forall F \in \mathcal{E} : \pi_F(t) \in R_F\}$$

We use n to denote the number of attributes $|\mathcal{V}|$ and m to denote the number of relations $|\mathcal{E}|$. IN denotes the sum of sizes of input relations in a query, and OUT denotes the output size.

⁴ We have been told that LogicBlox has implemented a similar algorithm recently, but their approach is not public. We shared our implementation with them several months ago.

Algorithm 1 Yannakakis($\mathcal{T} = (\mathcal{V}, \mathcal{E})$, $\{R_F | F \in \mathcal{V}\}$)

Input: Join tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, Relations R_F for each $F \in \mathcal{V}$

```

1: for all  $F \in \mathcal{V}$  in some bottom-up order do
2:    $P \leftarrow$  parent of  $F$ 
3:    $R_P \leftarrow R_P \bowtie R_F$ 
4: end for
5: for all  $F \in \mathcal{V}$  in some top-down order do
6:    $P \leftarrow$  parent of  $F$ 
7:    $R_F \leftarrow R_F \bowtie R_P$ 
8: end for
9: while  $F \in \mathcal{V}$  in some bottom-up order do
10:   $P \leftarrow$  parent of  $F$ 
11:   $R_P \leftarrow R_P \bowtie R_F$ 
12: end while
13: return  $R_R$  for the root  $R$ 

```

A *path* from $A \in \mathcal{V}_{\mathcal{H}}$ to $B \in \mathcal{V}_{\mathcal{H}}$ in a hypergraph \mathcal{H} is a sequence of attributes, starting with A and ending with B , such that each consecutive pair of attributes in the sequence occur together in a hyperedge. The number of attributes in the sequence is the *length* of the path.

We now define a GHD of a hypergraph.

- **Definition 2.**⁵ Given a hypergraph $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, a *generalized hypertree decomposition* is a pair (\mathcal{T}, χ) of a tree $\mathcal{T} = (\mathcal{V}_{\mathcal{T}}, \mathcal{E}_{\mathcal{T}})$ and function $\chi : \mathcal{V}_{\mathcal{T}} \rightarrow 2^{\mathcal{V}_{\mathcal{H}}}$ such that
 - For each relation $F \in \mathcal{E}_{\mathcal{H}}$, there exists a tree node $t \in \mathcal{V}_{\mathcal{T}}$ that covers the edge, i.e. $F \subseteq \chi(t)$.
 - For each attribute $A \in \mathcal{V}_{\mathcal{H}}$, the tree nodes containing A , i.e. $\{t \in \mathcal{V}_{\mathcal{T}} | A \in \chi(t)\}$, form a connected subtree.

The latter condition is called the “running intersection property”. The $\chi(t)$ sets are referred to as ‘bags’ of the GHD. GHDs are assumed to be ‘rooted’ trees, which imposes a top-down partial order on their nodes. Leveraging this order, for any GHD (\mathcal{T}, χ) and attribute $A \in \mathcal{V}_{\mathcal{H}}$, we define $TOP_{\mathcal{T}}(A)$ to be the top-most node $v \in \mathcal{V}_{\mathcal{T}}$ such that $A \in \chi(v)$.

When each bag of a GHD consists of the attributes of a single relation, the GHD is also called a *join tree*. Joins over a join tree can be processed using Yannakakis’ algorithm [28] (pseudo-code in Algorithm 1). The runtime of Yannakakis’ algorithm is $O(\text{IN} + \text{OUT})$.

GHDs can be interpreted as query plans for joins. Given a GHD, we first join the attributes in each bag using worst case optimal algorithms [19, 27] to get one intermediate relation per bag. The intermediate relations can then be joined using Yannakakis’ algorithm. This combined algorithm is called GHDJoin; Algorithm 3 in Appendix A gives the pseudo-code for GHDJoin.

The runtime of GHDJoin can be expressed in terms of the *fractional hypertree width* of the GHD:

- **Definition 3.** Given a hypergraph $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ and a GHD (\mathcal{T}, χ) , the *fractional hypertree width*, denoted $fhw(\mathcal{T}, \mathcal{H})$, is defined to be $\max_{t \in \mathcal{T}} \rho_t^*$ in which ρ_t^* is the optimal value of the

⁵ Traditionally GHDs are defined as a triple $(\mathcal{T}, \chi, \lambda)$ where the function $\lambda : \mathcal{V}_{\mathcal{T}} \rightarrow 2^{\mathcal{E}_{\mathcal{H}}}$ assigns relations to each bag. Here we omit this function and implicitly assign *every* relation to each bag (so $\lambda(t) = \mathcal{E}_{\mathcal{H}}$ for all $t \in \mathcal{V}_{\mathcal{T}}$). Though this makes a difference for certain notions of width, it leaves the fractional hypertree width unchanged, as adding more relations to the linear program will never make the objective value worse.

following linear program defined for each $t \in \mathcal{V}_{\mathcal{T}}$:

$$\begin{aligned} \text{Minimize } & \sum_{F \in \mathcal{E}_{\mathcal{H}}} x_F \log_{\text{IN}}(|R_F|) \text{ such that} \\ \forall A \in \chi(t) : & \sum_{F: A \in F} x_F \geq 1, \forall F \in \mathcal{E}_{\mathcal{H}} : x_F \geq 0 \end{aligned}$$

The fractional hypertree width is just the AGM bound [5] placed on the bags. Thus $\text{IN}^{fhw(\mathcal{T}, \mathcal{H})}$ is an upper bound on the sizes of the intermediate relations of GHDJoin. GHDJoin runs in time $\tilde{O}(\text{IN}^{fhw(\mathcal{T}, \mathcal{H})} + \text{OUT})$ for Join queries.

Annotated Relations

To define a general notion of aggregations, we look to relations annotated with semirings [11].

► **Definition 4.** A *commutative semiring* is a triple (S, \oplus, \otimes) of a set S and operators $\oplus : S \times S \rightarrow S$, $\otimes : S \times S \rightarrow S$ where there exist $0, 1 \in S$ such that for all $a, b, c \in S$ the following properties hold:

- Identity and Annihilation: $a \oplus 0 = a$, $a \otimes 1 = a$, $0 \otimes a = 0$
- Associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
- Commutativity: $a \oplus b = b \oplus a$, $a \otimes b = b \otimes a$
- Distributivity: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

Suppose we have some domain \mathbb{K} and an operator set $O = \{\oplus^1, \oplus^2, \dots, \oplus^k, \otimes\}$ such that 0 is the identity for each $\oplus^i \in O$ and $(\mathbb{K}, \oplus^i, \otimes)$ forms a commutative semiring for each i . We then define a relation with an annotation from \mathbb{K} for each tuple.

► **Definition 5.** An *annotated relation* with annotations from \mathbb{K} , or a \mathbb{K} -relation, over attribute set F is a set $\{(t_1, \lambda_1), (t_2, \lambda_2), \dots, (t_N, \lambda_N)\}$ such that for all $1 \leq i \leq N$, $t_i \in \mathcal{D}^F$, $\lambda_i \in \mathbb{K}$ and for all $1 \leq j \leq N : i \neq j \rightarrow t_i \neq t_j$.

Green et al. define a \mathbb{K} -relation to be a function $R_F : \mathcal{D}^F \rightarrow \mathbb{K}$ [11]. Our notion can be viewed as an explicit listing of this function's support. Note that unlike an explicit listing of the function's support, our table does allow tuples with 0 annotations. However, under our definitions of the operators below, an annotation of 0 is semantically equivalent to a tuple being absent (we discuss this further in Section 6). Note that we can have an annotated relation of the form R_{\emptyset} of size 1 containing the empty tuple with some annotation. We now define joins and aggregations over annotated relations.

Joins over Annotated Relations

Informally, a join over annotated relations is obtained as follows: (i) We perform a regular join on the non-annotated part of the relations. (ii) For each output tuple t of the join, we set its annotation to the product of the annotations of the input tuples used to produce t . We define a join $\bowtie_{F \in \mathcal{E}} R_F$ as:

$$\bowtie_{F \in \mathcal{E}} R_F = \{(t, \lambda) : \lambda = \prod_{F \in \mathcal{E}} \lambda_F \text{ in which } (\pi_F(t), \lambda_F) \in R_F\}$$

| R | | S | | $R \bowtie S$ | | | $\Sigma_C S$ | |
|-----|---|-----|---|---------------|---|---|--------------|---|
| A | B | B | C | A | B | C | B | K |
| 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 3 |
| 2 | 1 | 1 | 2 | 1 | 2 | 2 | 1 | 4 |
| | | | | 2 | 1 | 1 | 1 | 6 |
| | | | | 2 | 1 | 2 | 1 | 8 |

Figure 2 Selected examples illustrating the operators over the semiring $(\mathbf{R}_+, +, \cdot)$

Aggregations over Annotated Relations

An aggregation over an annotated relation R_F is specified by a pair (A, \oplus) where $A \in F$, and $\oplus \in O$. The aggregation takes groups of tuples in R_F that share values of all attributes other than A , and produces a single tuple corresponding to each group, whose annotation is the \oplus -aggregate of the annotations of the tuples in the group. Suppose that R has schema $R(A, B)$ in which A is a single attribute and B is a set of attributes. Then, the result of aggregation (A, \oplus) has only the attributes B and

$$\sum_{(A, \oplus)} R_{A, B} = \{(t_B, \lambda) : t_B \in \pi_B R \text{ and } \lambda = \sum_{(t, \lambda_t) \in R : \pi_B t = t_B}^{\oplus} \lambda_t\}$$

One can define the meaning of aggregate queries in a straightforward way: first compute the join and then perform aggregations. Figure 2 shows some examples of operators on relations. For the remainder of our work, we assume that all relations are \mathbb{K} -relations.

3.2 The AJAR problem

► **Definition 6.** Given some global attribute set \mathcal{V} and operator set O , we define an *aggregation ordering* to be a sequence $\alpha = \alpha_1, \alpha_2, \dots, \alpha_s$ such that for each $1 \leq i \leq s$, $\alpha_i = (a_i, \oplus_i)$ for some $a_i \in \mathcal{V}$, $\oplus_i \in O$ ⁶. In addition, attributes occur at most once, i.e., $a_j \neq a_k$ for each $1 \leq j < k \leq s$.

Informally, the aggregation ordering is just a sequence of attribute-operator pairs such that each attribute in the sequence occurs at most once. Note that the aggregation ordering specifies the order and manner in which attributes are aggregated. The ordering does not need to contain every attribute; we use the term *output attributes* to denote the attributes not in the ordering.

$V(\alpha)$ represents the set of attributes that appear in α , and $V(-\alpha)$ represents $V \setminus V(\alpha)$ (i.e. the output attributes). When $F \subseteq V(\alpha)$, we use α_F to represent a sequence β that is equivalent to α restricted to the attributes in F , i.e. $V(\beta) = F$, and any $(A, \oplus), (B, \oplus') \in \alpha$ such that $A, B \in F$ must also appear in β with their order preserved.

► **Definition 7 (AJAR).** Given some hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and an aggregation ordering α , an AJAR query $Q_{\mathcal{H}, \alpha}$ is a function over instances of \mathcal{H} such that

$$Q_{\mathcal{H}, \alpha}(\{R_F | F \in \mathcal{E}\}) = \Sigma_{\alpha_1} \cdots \Sigma_{\alpha_{|\alpha|}} \bowtie_{F \in \mathcal{E}} R_F.$$

For an AJAR query, we define OUT to be the final output size, rather than the output size of the join. There are two technical challenges when it comes to solving an AJAR query:

⁶ Note that, by this definition, the operators in aggregation ordering can be the product aggregation \otimes . However, product aggregations require different definitions, see Section 6.

- Multiple aggregation orders can give the same output over any database instance, and using some aggregation orders may give faster runtimes than others, e.g. some orders may allow early aggregation. Thus we need to identify which orders are equivalent to the given order and which order leads to the smallest runtime.
- OUT for an AJAR query with $|\alpha| > 0$ is smaller than the output size of the join part of the query. Thus the standard GHDJoin runtime of $\text{IN}^{fhw} + \text{OUT}$ is harder to achieve for AJAR queries. Naively applying a variant of GHDJoin that performs aggregations (Appendix Algorithm 5) to AJAR leads to a higher runtime of $\text{IN}^{fhw} \cdot \text{OUT}$ (see Appendix A). Thus we need to identify which GHDs can be used for efficient processing of AJAR queries.

We handle these technical challenges in turn.

3.3 Equivalent Orderings

Distinct aggregation orders can be equivalent in that they produce the same output on every instance. For example, suppose $\alpha = ((A, +), (B, +))$ and $\beta = ((B, +), (A, +))$, where A, B are two attributes in some \mathcal{H} . Then two AJAR queries with orderings α and β clearly produce the same output for any instance I over \mathcal{H} . This is because we can obtain β from α by switching the positions of two adjacent aggregations *with the same aggregation operator*. Similarly, if \mathcal{H} consists only of relations $\{A, B\}, \{B, C\}$, then the orderings $\alpha = ((A, +), (C, \max))$ and $\beta = ((C, \max), (A, +))$ are equivalent, since you can independently aggregate the two attributes away before joining the two relations on B . We now formally define equivalent orderings.

► **Definition 8** (Equivalent Orderings). Given a hypergraph \mathcal{H} , define the equivalence relation between orderings $\equiv_{\mathcal{H}}$ such that $\alpha \equiv_{\mathcal{H}} \beta$ if and only if $Q_{\mathcal{H}, \alpha}(I) = Q_{\mathcal{H}, \beta}(I)$ for all database instances I over the schema \mathcal{H} .

We say that two operators \oplus, \oplus' are distinct over a domain \mathbb{K} (denoted by $\oplus \neq \oplus'$) if $\exists x, y \in \mathbb{K} : x \oplus y \neq x \oplus' y$. And $\oplus = \oplus'$ means that $\forall x, y \in \mathbb{K}, x \oplus y = x \oplus' y$. Of course, distinct operators do not, in general, commute.

We now state a theorem specifying two conditions under which aggregations can commute. We will later show these conditions to be complete.

► **Theorem 9.** Suppose we are given a relation R_F such that $A, B \in F$ and two operators $\oplus', \oplus \in O$. Then

$$\Sigma_{(A, \oplus)} \Sigma_{(B, \oplus')} R_F = \Sigma_{(B, \oplus')} \Sigma_{(A, \oplus)} R_F$$

if one of the following conditions hold:

- $\oplus = \oplus'$
- There exist relations R_{F_1} and R_{F_2} such that $A \notin F_1, B \notin F_2$, and $R_{F_1} \bowtie R_{F_2} = R_F$.

Proof. The first condition follows trivially from the commutativity of our operators. The second condition follows from the fact that we can “push down” aggregations.

$$\begin{aligned} \Sigma_{(A, \oplus)} \Sigma_{(B, \oplus')} R_{F_1} \bowtie R_{F_2} &= (\Sigma_{(B, \oplus')} R_{F_1}) \bowtie (\Sigma_{(A, \oplus)} R_{F_2}) \\ &= \Sigma_{(B, \oplus')} \Sigma_{(A, \oplus)} R_{F_1} \bowtie R_{F_2} \end{aligned}$$

◀

These two conditions give us a simple procedure for testing when an ordering β is equivalent to the given α . Algorithm 2 gives the procedure’s pseudo-code. To avoid triviality, we assume α and β have the same set of attributes and assign the same operator to the same attributes. First we return true if both α and β are empty. Then we check if α can be shown to be equivalent to β using the conditions from Theorem 9. This procedure is both sound and complete:

Algorithm 2 TestEquivalence($\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, α, β)

Input: Query hypergraph \mathcal{H} , orderings α, β .**Output:** True if $\alpha \equiv_{\mathcal{H}} \beta$, False otherwise.

```

if  $|\alpha| = |\beta| = 0$  then
    return True
end if
Remove  $V(-\alpha)$  from  $\mathcal{H}$ , then divide  $\mathcal{H}$  into connected components  $C_1, \dots, C_m$ .
if  $m > 1$  then
    return  $\wedge_i \text{TestEquivalence}(\mathcal{H}, \alpha_{C_i}, \beta_{C_i})$ 
end if
Choose  $j$  such that  $\beta_j = \alpha_1$ . Let  $\beta_j = (b_j, \oplus'_j)$ .
if  $\exists i < j : \beta_i = (b_i, \oplus'_i)$ ,  $\oplus'_i \neq \oplus'_j$  and there is a path from  $b_i$  to  $b_j$  in  $\{b_i, b_{i+1}, \dots, b_{|\alpha|}\}$ 
then
    return False
end if
Let  $\beta'$  be  $\beta$  with  $\beta_j$  removed.
Let  $\alpha'$  be  $\alpha$  with  $\alpha_1$  removed.
return TestEquivalence( $\mathcal{H}, \alpha', \beta'$ )

```

► **Lemma 10.** *Algorithm 2 returns True iff $\alpha \equiv_{\mathcal{H}} \beta$.*

We omit this lemma's proof because it is very similar to and implied by the proofs required in Section 4.

To answer AJAR queries, we need one more component in addition to Algorithm 2; namely AggroGHDJoin, a straightforward variant of GHDJoin that performs aggregations (Algorithm 5 in Appendix A). The first step of AggroGHDJoin is similar to that of GHDJoin, namely performing joins within each bag of the GHD to get intermediate relations. We need to do some extra work to ensure that each annotation is multiplied only once, since a relation may be joined in multiple bags. After that, instead of calling Yannakakis' algorithm on the intermediate relations, AggroGHDJoin calls AggroYannakakis (Algorithm 4 in Appendix A), a well-known variant of Yannakakis that performs aggregations. AggroYannakakis initially performs semijoins like Yannakakis (lines 1-8 in Algorithm 1). But in the bottom-up join phase (line 11), AggroYannakakis aggregates out all attributes that have F as their TOP node, before joining R_F with R_P .

Armed with Algorithm 2 and AggroGHDJoin, we have a simple way to answer an AJAR query $Q_{\mathcal{H}, \alpha}$. We search through all orders, running Procedure 1 to check for equivalence with α . For each order β such that $\beta \equiv_{\mathcal{H}} \alpha$, we search all through GHDs and check if they are compatible with β . A GHD \mathcal{T} is defined to be *compatible* with an ordering β if, for all attribute pairs A, B , $TOP_{\mathcal{T}}(A)$ being an ancestor of $TOP_{\mathcal{T}}(B)$ implies that either A is an output variable or A occurs before B in β (note this precludes B from being an output variable). We can run AggroGHDJoin on any compatible GHD to answer the AJAR query. The runtime of AggroGHDJoin on a compatible GHD (\mathcal{T}, χ) is given by $\tilde{O}(\text{IN}^{fhw(\mathcal{T}, \mathcal{H})} + \text{OUT})$. We choose the compatible GHD that has the smallest fhw , and use it to answer the query. The theorem below states our runtime:

► **Theorem 11.** *Given a AJAR query $Q_{\mathcal{H}, \alpha}$, let w^* denote the smallest fhw for a GHD compatible with an ordering $\equiv_{\mathcal{H}} \alpha$; the runtime of our approach is $\tilde{O}(\text{IN}^{w^*} + \text{OUT})$.*

Comparison to Prior Work

Work by Olteanu and Zavodny [6, 21] focuses on a special case of AJAR queries, having a single aggregation operator. For these queries, they have a similar algorithm that iterates over

GHDs to find the best compatible one. Their algorithm achieves the same runtime as ours, but cannot handle queries with more than one type of aggregation operator. The FAQ paper uses an algorithm called InsideOut to answer general AJAR queries. The running time of InsideOut equals $\tilde{O}(\text{IN}^{\text{faqw}})$ where faqw (FAQ-width) is a new notion of width defined by the FAQ authors [15, Section 9.1]. Our algorithm has runtime that is no worse than InsideOut ($w^* \leq \text{faqw}$, $\text{OUT} \leq \text{IN}^{\text{faqw}}$), and can be much better when output attributes are present.

► **Theorem 12.** *For any AJAR query, $w^* \leq \text{faqw}$ and $\text{OUT} \leq O(\text{IN}^{\text{faqw}})$.*

This theorem is proved in Appendix B.1. Notice that the InsideOut runtime is not *output-sensitive*, i.e. it does not have a $+$ OUT term. As a result the runtime can be very high when the output is small relative to the number of output attributes; this is demonstrated by Example 58 in the appendix. FAQ does have a high-level discussion of approaches to make InsideOut output-sensitive [15, Section 10.2]; indeed, simply using GHDJoin instead of their bespoke algorithm can achieve output-sensitive bounds, which we discuss in Appendix B.

Discussion

We presented a remarkably simple procedure for solving AJAR queries. The procedure involves a brute force search over different orderings and GHDs, but this is usually unavoidable as finding the best ordering and GHD is NP-Hard. Deciding if an ordering is equivalent to the given ordering is enabled by Algorithm 2, which takes time polynomial in the number of attributes. Determining if a GHD is compatible with an ordering is straightforward as well. Once the best GHD is found, we use well known, standard algorithms like AggroGHDJoin to answer the query efficiently. The resulting runtime exponents are smaller than those of previous work. The simplicity of the algorithm makes it easy to implement; we have already implemented a special case of a single additive operator \oplus in our engine [1].

The equivalence/compatibility tests raise the technically interesting question of finding succinct characterizations of:

- All orderings equivalent to any given α .
- All GHDs that are compatible with at least one of the equivalent orderings.

We answer the first question in Section 4 by providing a simple characterization of all equivalent orderings, and the second question in Section 5 by defining ‘valid’ GHDs and characterizing their structure in relation to unrestricted GHDs.

4 Characterizing Equivalent Orderings

We described a procedure for determining when two orderings are equivalent. The equivalence relation $\equiv_{\mathcal{H}}$ defines equivalence classes among the orderings, but these classes may be exponential in size; we find a more succinct characterization that lets us enumerate all equivalent orderings. Chen and Dalmau [7] obtained a similar order-equivalence characterization for a special case of the AJAR problem, namely for aggregations “and” and “or”. The characterization was based on a procedure that generated all equivalent orderings. We improve on this result by providing a simple and succinct characterization of the equivalence class of an aggregation ordering with any number operators.

To that end, we develop an enumeration of the constraints that are sufficient and necessary for an ordering to be in the equivalence class of α . The constraints are of the form “ A must always occur before B ”:

► **Definition 13 (PREC).** Given an AJAR query $Q_{\mathcal{H}, \alpha}$, define a constraint $\text{PREC} \subseteq \mathcal{V} \times \mathcal{V}$ such that $(A, B) \in \text{PREC}$ if and only if A precedes B in all orderings that are equivalent to α .

We say $\text{PREC}(A, B)$ is true if and only if $(A, B) \in \text{PREC}$.

Trivially, the number of pairs in PREC is less than n^2 . We note that we can use PREC to define a (strict) partial ordering on the attributes; the constraints are clearly antireflexive, antisymmetric, and transitive. We use $<_{\mathcal{H}, \alpha}$ to denote this partial order. Given an AJAR query $Q_{\mathcal{H}, \alpha}$, $<_{\mathcal{H}, \alpha}$ is a partial order of attribute-operator pairs such that for any $(A, \oplus), (B, \oplus') \in \alpha$, $(A, \oplus) <_{\mathcal{H}, \alpha} (B, \oplus')$ if $\text{PREC}(A, B)$ (see Definition 22 for the exact definition). The partial order $<_{\mathcal{H}, \alpha}$ is easier to use for proofs; we use the partial order to show the soundness and completeness of these constraints.

► **Theorem 14** (Soundness and Completeness of $<_{\mathcal{H}, \alpha}$). *Suppose we are given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and aggregation orderings α, β . Then $\alpha \equiv_{\mathcal{H}} \beta$ if and only if β is a linear extension of $<_{\mathcal{H}, \alpha}$.*

We first describe a procedure to compute the precedence relation PREC . After that, we reason about its completeness.

Computing PREC

To assist in building PREC , we define a constraint of the form ‘ A and B cannot commute’:

► **Definition 15** (DNC). Given an AJAR query $Q_{\mathcal{H}, \alpha}$, define a constraint $\text{DNC} \subseteq \mathcal{V} \times \mathcal{V}$ such that $(A, B) \in \text{DNC}$ if and only if A and B are in the same order in any β such that $\beta \equiv_{\mathcal{H}} \alpha$.

Once again, we say $\text{DNC}(A, B)$ is true if and only if $(A, B) \in \text{DNC}$. We prefer to work with DNC because we have already discussed when aggregations can commute in Theorem 9; the conditions of that theorem specify when DNC is *FALSE*. However, we can immediately derive a simple relationship between PREC and DNC :

► **Lemma 16.** *Given an AJAR query $Q_{\mathcal{H}, \alpha}$, for any $A, B \in \mathcal{V}$, $\text{PREC}(A, B)$ iff $\text{DNC}(A, B)$ and A precedes B in α .*

We now develop conditions when DNC is true. Recall that Theorem 9 states that two aggregations can commute if (1) they have the same operator or (2) if they can be separated in the join query; the simplest structure that violates both of these conditions is an edge that contains two attributes with differing aggregating operators.

► **Lemma 17.** *Given an AJAR query $Q_{\mathcal{H}, \alpha}$, suppose $(A, \oplus), (B, \oplus') \in \alpha$. If $\oplus \neq \oplus'$ and there exists an edge $E \in \mathcal{E}$ such that $A, B \in E$, then $\text{DNC}(A, B)$.*

Lemma 17 serves as a base case, but we want to extend the violation of Theorem 9’s conditions beyond single edges to paths. To do so, consider the following examples of how our commuting conditions interact with paths of length two.

► **Example 18.** Consider the query

$$\sum_A \max_B \max_C R(A, B) \bowtie S(B, C) \text{ hence } \alpha = (A, B, C).$$

No two attributes can be separated, which implies $\text{DNC}(A, B)$ and $\text{DNC}(A, C)$. Lemma 17 gives us the former constraint, but not the latter one. This example indicates that it may be possible to extend a constraint $\text{DNC}(A, B)$ along an edge $\{B, C\}$. On the other hand, consider the query

$$\max_B \sum_A \max_C R(A, B) \bowtie S(B, C) \text{ so } \alpha = (B, A, C).$$

Note that A and C can be separated, which implies that only $\text{DNC}(A, B)$ holds. Note that, as before, Lemma 17 gives us this constraint. This example suggests that we cannot extend every $\text{DNC}(A, B)$ constraint along an additional edge.

The key difference between the two examples is the *relative order* of A and B in α , which suggests that we can only extend $\text{DNC}(A, B)$ along an edge if A precedes B in α , i.e. if $\text{PREC}(A, B)$.

► **Lemma 19.** *Given an AJAR query $Q_{\mathcal{H}, \alpha}$, suppose $(A, \oplus), (B, \oplus') \in \alpha$. If $\oplus \neq \oplus'$ and $\exists C \in \mathcal{V}, E \in \mathcal{E} : \text{PREC}(A, C)$ and $B, C \in E$, then $\text{DNC}(A, B)$.*

PREC is transitive, which implies:

► **Lemma 20.** *Given an AJAR query $Q_{\mathcal{H}, \alpha}$, suppose $(A, \oplus), (B, \oplus') \in \alpha$. If $\exists C : \text{PREC}(A, C)$ and $\text{PREC}(C, B)$, then $\text{DNC}(A, B)$.*

The above transitivity condition interacts with the condition from Lemma 19 in interesting ways.

► **Example 21.** Consider the query with $\alpha = (A, B, C, D)$,

$$\sum_A \max_B \max_C \sum_D R(A, B) \bowtie S(B, D) \bowtie T(C, D).$$

No attributes can be separated, which implies $\text{DNC}(A, B)$, $\text{DNC}(A, C)$, $\text{DNC}(B, D)$, and $\text{DNC}(C, D)$. Transitivity gives $\text{DNC}(A, D)$ as well. Now let us derive these constraints using Lemmas 17, 19, and 20. Lemma 17 gives us $\text{DNC}(A, C)$, $\text{DNC}(B, D)$, and $\text{DNC}(C, D)$. Note that at this point, Lemma 19 gives us no more constraints. Only after the transitivity of Lemma 20 adds the constraint $\text{DNC}(A, D)$ can Lemma 19 add the constraint $\text{DNC}(A, B)$, completing the set of constraints.

It turns out that these three relatively simple lemmas are the sufficient and necessary constraints on the equivalence classes of orderings; no other conditions are necessary to complete the proofs the soundness and completeness of $<_{\mathcal{H}, \alpha}$.

We note that our current specifications of PREC and DNC are mutually recursive. The PREC and DNC sets build up in rounds; Lemma 17 provides their initial values, and Lemmas 16, 19, and 20 iteratively build up the sets further. We keep applying these lemmas until the sets reach a fixed point. This takes at most $2|\alpha|^2$ iterations, as we must add at least one additional attribute pair per iteration, and there can be only $|\alpha|^2$ pairs of attributes in each set. Thus the overall runtime of computing these constraints is polynomial in the number of attributes. We detail this process in Appendix C.

For convenience of notation, we make one modification to the definition of the partial order $<_{\mathcal{H}, \alpha}$. When A is an output attribute and B is not, we define $A <_{\mathcal{H}, \alpha} B$ to be true. So we can formally state the definition as:

► **Definition 22** ($<_{\mathcal{H}, \alpha}$). Given a AJAR query $Q_{\mathcal{H}, \alpha}$, we define $A <_{\mathcal{H}, \alpha} B$ to be true if either (i) A is an output attribute and B is not, or (ii) $\text{PREC}(A, B)$ is true.

Soundness and Completeness of $<_{\mathcal{H}, \alpha}$

To give an intuition on how we prove the soundness and completeness of $<_{\mathcal{H}, \alpha}$, we now state two key lemmas (with proofs in Appendix C) illustrating properties of $<_{\mathcal{H}, \alpha}$.

► **Lemma 23.** *Suppose we are given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and an aggregation ordering α . Suppose $(A, \oplus), (B, \oplus') \in \alpha$ for differing operators $\oplus \neq \oplus'$. Then, for any path P in \mathcal{H} between A and B , there must exist some attribute in the path $C \in P$ such that $C <_{\mathcal{H}, \alpha} A$ or $C <_{\mathcal{H}, \alpha} B$.*

Lemma 23 intuitively states that incomparable attributes with different operators must be separated in \mathcal{H} by their common predecessors in $<_{\mathcal{H}, \alpha}$.

- **Lemma 24.** *Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and an aggregation ordering α , suppose we have two attributes $A, B \in V(\alpha)$ such that $A <_{\mathcal{H}, \alpha} B$. Then there must exist a path P from A to B such that for every $C \in P, C \neq A$ we have $A <_{\mathcal{H}, \alpha} C$.*

Given these two lemmas, the proof of Theorem 14 is straightforward. Lemma 23 implies that, given an attribute ordering β that is a linear extension of $<_{\mathcal{H}, \alpha}$, each inversion of attribute-operator pairs must either have equal operators or have attributes that can be separated, allowing us to repeatedly use Theorem 9 to transform β into α . Lemma 24 implies that, given an attribute ordering β that is not a linear extension of $<_{\mathcal{H}, \alpha}$, we can construct a counterexample.

Discussion

We obtained a sound and complete characterization of all orderings equivalent to any given ordering. This result extends the work of Chen and Dalmau [7], who had characterized equivalent orderings for queries with logical “and” and “or” operators. Our characterization is simple, consisting of a partial order whose linear extensions are precisely the equivalent orderings. FAQ [15]’s method for identifying equivalent orderings is sound but not complete. That is, there exist equivalent orderings that the FAQ method does not identify as being equivalent (Appendix Example 59). In contrast, our characterization is guaranteed to cover all valid orderings. This completeness property lets us create a decomposition that is guaranteed to preserve all *node-monotone* widths (see Definition 28). This in turn lets us get tighter guarantees on our runtime exponent, using the notion of submodular width (Section 5.3).

5 Decomposing Valid GHDs

We express our AJAR algorithm directly in terms of GHDs, rather than in terms of aggregation orderings. As such, our goal is the characterization of GHDs that are compatible with at least one equivalent ordering, i.e. the GHDs that can be used to answer an AJAR query. We call a GHD *valid* if it is compatible with at least one equivalent ordering. We first give a simple characterization of valid GHDs. Then we demonstrate a way to reduce the problem of finding a minimum-width valid GHD to multiple subproblems on unconstrained GHDs (Section 5.1). This decomposition of the problem gets us three things:

- We can speed up our brute force search for an optimal valid GHD. We can also find approximately optimal valid GHDs in polynomial time using Marx’s GHD approximation algorithm [16] (Section 5.2).
- We can apply existing MapReduce join algorithms that utilize GHDs [3], obtaining efficient parallel algorithms for solving AJAR queries (Section 5.4).
- We can apply improved join algorithms [13, 17] to further reduce our runtime exponent (Section 5.3).

5.1 Valid and Decomposable GHDs

We can easily characterize valid GHDs by combining the definition of compatible GHDs with Theorem 14.

- **Theorem 25.** *For a AJAR query $Q_{\mathcal{H}, \alpha}$, a GHD (\mathcal{T}, χ) is valid if and only if for every pair of attributes A, B such that $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, $B \not<_{\mathcal{H}, \alpha} A$.*

Theorem 25 gives us a criterion specifying which GHDs can act as query plans. We now consider the problem of finding a minimum width valid GHD for any AJAR query. We call a GHD *optimal* if it has the minimum width possible for valid GHDs. We show how to reduce

the problem of finding an optimal valid GHD into smaller problems of finding ordinary optimal GHDs. This unlocks a trove of powerful GHD results and makes them applicable to our problem.

Given an AJAR query $Q_{\mathcal{H},\alpha}$, suppose we have a subset of the nodes $V \subseteq \mathcal{V}$. Define \mathcal{E}_V to be $\{E \in \mathcal{E} | E \cap V \neq \emptyset\}$, i.e. the set of edges that intersect with V . As before, α_V denotes the aggregation ordering restricted to the nodes in V . Additionally define V^O to be $\{v \in V | \forall w \in V, w \not\prec_{\mathcal{H},\alpha} v\}$, i.e. the nodes in V that have no predecessors in V according to the partial ordering $\prec_{\mathcal{H},\alpha}$. Finally, note that $\alpha_{V \setminus V^O}$ is then α_V with all the nodes in V^O removed (note that this makes the nodes in V^O output attributes).

► **Definition 26.** Given an AJAR query $Q_{\mathcal{H},\alpha}$, we say a GHD (\mathcal{T}, χ) is *decomposable* if:

- There exists a rooted subtree \mathcal{T}_0 of \mathcal{T} such that $\chi(\mathcal{T}_0) = \mathcal{V}(-\alpha)$ (i.e. output attributes).
- For each connected component C of $\mathcal{H} \setminus V_{-\alpha}$, there is exactly one subtree $\mathcal{T}_C \in \mathcal{T} \setminus \mathcal{T}_0$ such that \mathcal{T}_C is a decomposable GHD of $Q_{(\cup_{E \in \mathcal{E}_C} E, \mathcal{E}_C), \alpha_{C \setminus C^O}}$.

We start by connecting this idea of decomposable GHDs to valid GHDs. We only give proof sketches here; see appendix D for the full proofs.

► **Theorem 27.** *Every decomposable GHD is valid.*

Proof. (Sketch) Suppose the AJAR query is $Q_{\mathcal{H},\alpha}$. We need to show for any A, B such that $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, $A \not\prec_{\mathcal{H},\alpha} B$. We use induction on $|\alpha|$. If $|\alpha| = 0$, all GHDs are valid and decomposable. For $|\alpha| > 0$, \mathcal{T}_0 ensures that the output attributes are above non-output attributes. If A and B are non-output attributes and $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, then both are in some \mathcal{T}_C . By the inductive hypothesis, \mathcal{T}_C is valid with respect to $Q_{(\cup_{E \in \mathcal{E}_C} E, \mathcal{E}_C), \alpha_{C \setminus C^O}}$. By inspecting the partial order created by this subgraph, we conclude that $A \not\prec_{\mathcal{H},\alpha} B$ as desired. ◀

Every valid GHD may not be decomposable. However, every valid GHD can be transformed into a corresponding decomposable GHD using some simple transformations. Each bag of the resulting decomposable GHD is a subset of one of the bags of the original GHD. Thus the fhw of the decomposable GHD is at most the fhw of the original valid GHD. In fact, we can make a more general claim, using a notion of node-monotone functions, defined next.

► **Definition 28.** Given a hypergraph $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, we define a function to be *node-monotone* if it is a function $\gamma : 2^{\mathcal{V}_{\mathcal{H}}} \rightarrow \mathbb{R}$ such that $\forall A \subseteq B \subseteq \mathcal{V}_{\mathcal{H}} : \gamma(A) \leq \gamma(B)$. Given any node-monotone function γ , we define the γ -width of a GHD (\mathcal{T}, χ) over \mathcal{H} as $\max_{v \in \mathcal{V}_{\mathcal{T}}} \gamma(\chi(v))$.

Many standard notions of widths can be expressed as γ -widths for a suitably chosen γ . Specifically:

► **Proposition 1.** Suppose we are given a hypergraph $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ and database instance I on \mathcal{H} . Then for each of following notions of width: (i) Treewidth (ii) Generalized Hypertree Width (iii) Fractional Hypertree Width (iv) Submodular Width, there exists a node-monotone function γ such that γ -width equals the given notion of width.

As a simple example, tree-width can be expressed as γ -width for $\gamma(A) = |A| - 1$. We can now relate valid and decomposable GHDs with respect to their γ -widths.

► **Theorem 29.** *For every valid GHD (\mathcal{T}, χ) , there exists a decomposable GHD (\mathcal{T}', χ') such that for all node-monotone functions γ , the γ -width of (\mathcal{T}', χ') is no larger than the γ -width of (\mathcal{T}, χ) .*

Proof. (Sketch) Suppose the AJAR query is $Q_{\mathcal{H},\alpha}$. We transform the given GHD (\mathcal{T}, χ) into (\mathcal{T}', χ') such that for each $v' \in \mathcal{V}'_{\mathcal{T}}$, there exists a $v \in \mathcal{V}_{\mathcal{T}}$ such that $\chi'(v') \subseteq \chi(v)$. The result then follows from the node-monotonicity of γ and the definition of γ -width. Any transformation of a GHD that ensures that all new bags are subsets of old bags, is called width-preserving.

We then transform the GHD (\mathcal{T}, χ) to satisfy the following properties (using width-preserving transformations):

- Every $t \in \mathcal{T}$ is $TOP_{\mathcal{T}}(A)$ for exactly one attribute A .
- For any node $t \in \mathcal{T}$ and the subtree \mathcal{T}_t rooted at t , the attributes $\{v \in \mathcal{V} | TOP_{\mathcal{T}}(v) \in \mathcal{T}_t\}$ form a connected subgraph of \mathcal{H} .

We can show, by induction, any valid GHD that satisfies these two properties is decomposable. Intuitively, the first transformation ensures the subtree \mathcal{T}_0 exists as desired. The second transformation ensures that each of the \mathcal{T}_C 's exists and satisfies the requisite properties. \blacktriangleleft

This theorem lets us restrict our search to the smaller space of decomposable GHDs (instead of all valid GHDs) when looking for the optimal valid GHD. Moreover, the space of decomposable GHDs is simpler; it can be factored into smaller spaces of unconstrained GHDs, as we show next. We present the definition of *characteristic hypergraphs*, which are intuitively the set of hypergraphs that specify the factors, i.e. the unconstrained GHDs.

Our goal is two-fold: (1) to be able to split a decomposable GHD into component GHDs of the characteristic hypergraphs and (2) to be able to take arbitrary GHDs of the characteristic hypergraphs and connect them to create a decomposable GHD of the original AJAR problem. The definition of decomposable GHDs decomposes a GHD into a series of sub-trees $\mathcal{T}_0, \dots, \mathcal{T}_k$. The definition specifies that the subtrees $\mathcal{T}_1, \dots, \mathcal{T}_k$ must be decomposable GHDs of (smaller) AJAR problems. Additionally, it is simple to show \mathcal{T}_0 is a GHD of the hypergraph $(V(-\alpha), \{E \in \mathcal{E} | E \subseteq V(-\alpha)\})$. If we apply this decomposition recursively to the subtrees $\mathcal{T}_1, \dots, \mathcal{T}_k$, we can divide any decomposable GHD into a series of (unrestricted) GHDs of particular hypergraphs. This provides the basis of our definition of the characteristic hypergraphs; we define a hypergraph \mathcal{H}_0 that specifies the hypergraph corresponding to \mathcal{T}_0 and then recurse on the smaller AJAR queries specified in Appendix Definition 85.

However, if we are given arbitrary GHDs of the hypergraphs as defined thus far, we may not be able to stitch them together while preserving the running intersection property of GHDs. To ensure this stitching is possible, we need the characteristic hypergraphs to contain additional edges that we can use to guarantee the running intersection property. Intuitively the edges we add will be the intersections of the adjacent subtrees in our decomposition; for example, for any connected component C of $\mathcal{H} \setminus V(-\alpha)$, \mathcal{T}_0 and \mathcal{T}_C are adjacent, and we will add the edge $\chi(\mathcal{T}_0) \cap \chi(\mathcal{T}_C)$ to the corresponding hypergraphs. We can use these ‘intersection edges’ to connect particular nodes in the adjacent subtrees.

► **Definition 30.** Given an AJAR problem $Q_{\mathcal{H}, \alpha}$, suppose C_1, \dots, C_k are the connected components of $\mathcal{H} \setminus V_{-\alpha}$. Define a function $H(\mathcal{H}, \alpha)$ that maps AJAR queries to a set of hypergraphs as follows:

- $C_i^+ = \bigcup_{E \in \mathcal{E}_{C_i}} E$ for all $1 \leq i \leq k$
- $\mathcal{H}_0 = (V_{-\alpha}, \{F \in \mathcal{E} | F \subseteq V_{-\alpha}\} \cup \{V_{-\alpha} \cap C_i^+ | 1 \leq i \leq k\})$
- $\mathcal{H}_i^+ = (C_i^+, \mathcal{E}_{C_i} \cup \{V_{-\alpha} \cap C_i^+\})$
- $H(\mathcal{H}, \alpha) = \{\mathcal{H}_0\} \cup \bigcup_{1 \leq i \leq k} H(\mathcal{H}_i^+, \alpha_{C_i \setminus C_i^+})$

The hypergraphs in the set $H(\mathcal{H}, \alpha)$ are defined to be the *characteristic hypergraphs*.

Note that the definition of characteristic hypergraphs depends only on (\mathcal{H}, α) , and not on a specific GHD or the instance. Now we state a key result that lets us reduce the problem of searching for an optimal *valid* GHD over \mathcal{H} to that of searching for (not necessarily valid) optimal GHDs over characteristic hypergraphs. Each decomposable GHD corresponds to a GHD over each characteristic hypergraph; conversely, a combination of GHDs for characteristic hypergraphs gives us a decomposable GHD for \mathcal{H} . Formally:

► **Theorem 31.** For an AJAR query $Q_{\mathcal{H}, \alpha}$, suppose $\mathcal{H}_0, \dots, \mathcal{H}_k$ are the characteristic hypergraphs $H(\mathcal{H}, \alpha)$. Then GHDs G_0, G_1, \dots, G_k of $\mathcal{H}_0, \dots, \mathcal{H}_k$ can be connected to form a decomposable GHD G for $Q_{\mathcal{H}, \alpha}$. Conversely, any decomposable GHD G of $Q_{\mathcal{H}, \alpha}$ can be partitioned into GHDs

G_0, G_1, \dots, G_k of the characteristic hypergraphs $\mathcal{H}_0, \dots, \mathcal{H}_k$. Moreover, in both of these cases, $\gamma\text{-width}(G) = \max_i \gamma\text{-width}(G_i)$.

The proof is provided in the appendix, but it is a straightforward application of definitions.

► **Corollary 32.** *Given an optimal GHD for each characteristic hypergraph of an AJAR query $Q_{\mathcal{H}, \alpha}$, we can construct an optimal valid GHD. The width of the optimal valid GHD equals the maximum optimal-GHD-width over its characteristic hypergraphs.*

This reduces the problem of finding the optimal valid GHD to smaller problems of finding optimal GHDs. We first present the decomposition in the FAQ [15] paper. Then we present several applications of our decomposition, and compare them to their FAQ analogues.

FAQ's Decomposition

The FAQ paper uses a decomposition of the problem that is not width-preserving. They remove the set of output attributes $V(-\alpha)$ and decompose the rest of the hypergraph into smaller hypergraphs. They construct a regular Variable-Ordering/GHD for each hypergraph. Then they add all output attributes $V(-\alpha)$ into each bag of each of the GHDs, and then stitch the GHDs together. This *output addition* to the bags of the GHDs leads to a potentially $2 \times$ increase in width compared to our method which stitches the GHDs together without changing their width. As a result, FAQ's decomposition incurs higher runtime costs in each application of the decomposition, as we see in the next three subsections.

► **Example 33.** Consider a query with output attribute A

$$\sum_{B,+} \sum_{C,+} (R(A, B) \bowtie S(B, C)).$$

The optimal valid GHD for this query has bags $\{A, B\}$ and $\{B, C\}$, and thus has fhw 1. The faqw is also 1. If we apply our decomposition, we get a GHD with bags $\{A\}$, $\{A, B\}$, $\{B, C\}$ which still has fhw 1. FAQ's decomposition on the reduced hypergraph (with output attribute A removed) has one bag $\{B, C\}$. Adding A to it gives a single bag $\{A, B, C\}$ resulting in a fhw of 2. More generally, consider query Q_n with $\alpha = ((B_1, +), (B_2, +), \dots, (B_n, +))$ and relations $T(A_1, B_1)$ and also $R_{i,j}(A_i, A_j)$, $S_{i,j}(B_i, B_j)$ for $i, j \in \{1, 2, \dots, n\}$. Our decomposition gives a GHD with bags $\{A_1, A_2, \dots, A_n\}$, $\{A_1, B_1\}$, $\{B_1, B_2, \dots, B_n\}$, which has fhw $n/2$. FAQ's decomposition has a single bag and fhw equal to n .

5.2 Finding optimal valid GHDs

Armed with Corollary 32, we simplify the brute force search algorithm for finding optimal valid GHDs.

► **Theorem 34.** *Let $Q_{\mathcal{H}, \alpha}$ be an AJAR query. The optimal width valid GHD for this query can be found in time $\tilde{O}(|\mathcal{H}|2^{\tilde{O}(\max_{\mathcal{H}' \in H(\mathcal{H}, \alpha)}(|\mathcal{H}'|))})$.*

This runtime for finding the optimal valid GHD can be exponentially better than the naive runtime:

► **Example 35.** Consider the star query $\mathcal{H} = (\{A, B_1, \dots, B_n\}, \{\{A, B_i\} \mid 1 \leq i \leq n\})$, $\alpha = (B_1, +), (B_2, +), \dots, (B_n, +)$. A is the only output attribute. Removing A breaks the hypergraph into n components, so there are $n + 1$ characteristic hypergraphs, each of size ≤ 2 . Finding the optimal valid GHD takes time $\tilde{O}(n)$, whereas the standard algorithm takes time exponential in n .

We can also approximate the GHD [16]:

► **Theorem 36** (Marx’s GHD approximation). *Let Q be a join query with hypergraph \mathcal{H} and fractional hypertree width w . Then we can find a GHD for Q in time polynomial in $|\mathcal{H}|$, that has width $w' \leq w^3$.*

We can replicate Marx’s result for valid GHDs.

► **Theorem 37.** *Let $Q_{\mathcal{H},\alpha}$ be an AJAR query, such that its minimum width valid GHD has width w . Then we can find a valid GHD in time polynomial in $|\mathcal{H}|$ that has width $w' \leq w^3$.*

FAQ [15]’s decomposition lets them apply Marx’s approximation as well. However, their decomposition is not width-preserving i.e. the width of their final GHD is higher than the width of the GHDs they construct for the hypergraphs in the decomposition. Thus their decomposition gives a weaker width guarantee of $faqw^3 + faqw$ [15, Theorem 9.49]. The extra $+faqw$ factor is due to output addition. Our guarantee, w^3 , is strictly smaller (w is the width of the optimal valid GHD) as $w \leq faqw$ by Theorem 12.

5.3 Tighter Runtime Exponents

Marx [17] introduced the notion of submodular width (sw) that is tighter than fhw , and showed that a join query can be answered in time $\text{IN}^{O(sw)}$. The O in the exponent is because Marx’s algorithm requires expensive preprocessing that takes $\text{IN}^{2 \times sw}$ time. After the pre-processing, the join can be performed in time IN^{sw} . Despite the O in the exponent, this algorithm can be very valuable because there are families of hypergraphs that have unbounded fhw but bounded sw . We can apply Marx’s algorithm to the characteristic hypergraphs, potentially improving our runtime. Marx also showed that joins on a family of hypergraphs are fixed parameter tractable if and only if the submodular width of the hypergraph family is bounded [17]. Moreover, adaptive width [18] (applicable only when relations are expressed as truth tables) is unbounded for a hypergraph family if and only if submodular width is unbounded. Corollary 32 gets us an analogous tractability result for AJAR queries.

► **Theorem 38.** *We can answer an AJAR query $Q_{\mathcal{H},\alpha}$ in time $O(\text{IN}^{O(\max_{\mathcal{H}' \in H(\mathcal{H},\alpha)}(sw(\mathcal{H}')))} + \text{OUT})$.*

Recent work [13] uses degree information to more tightly bound the output size of a query. The bound in the reference, called the *DBP bound*, has a tighter exponent than the AGM bound, while requiring only linear preprocessing to obtain. The authors also provide algorithms whose runtime matches the DBP bound. We can define DBP-width $dbpw(\mathcal{T}, \mathcal{H})$ such that $\text{IN}^{dbpw(\mathcal{T}, \mathcal{H})}$ is the maximum value of the DBP bound over all bags of GHD \mathcal{T} . We then use the improved algorithm in place of GJ in AggroGHDJoin. This lets us get tighter results “for free”, reducing our runtime to IN^{dbpw} instead of IN^{fhw} . Formally:

► **Theorem 39.** *Given an AJAR query $Q_{\mathcal{H},\alpha}$ and a valid GHD for \mathcal{H} , we can answer the query in time $O(\text{IN}^{dbpw(\mathcal{T}, \mathcal{H})} + \text{OUT})$. Equivalently, we can answer the query in time $O(\text{IN}^{\max_{\mathcal{H}' \in H(\mathcal{H},\alpha)} dbpw(\mathcal{T}, \mathcal{H}')} + \text{OUT})$.*

As discussed before, FAQ has a non-width-preserving decomposition. We can combine FAQ’s decomposition with the DBP bound as we did above. Suppose we perform FAQ’s decomposition, and IN^{faqw+} denotes the highest value of the DBP bound on each of their characteristic hypergraphs, and on the set of output attributes. Thus the DBP-width of each of their characteristic hypergraphs, and the outputs, is $faqw+$. However, when they perform output addition, the DBP-width of the resulting GHDs can go up to $2faqw+$. This happens when the DBP bound on both the outputs and one of the characteristic hypergraphs equals IN^{faqw+} . So if we apply

the DBP result to FAQ’s decomposition, we get a runtime of $\tilde{O}(\text{IN}^{2faqw} + \text{OUT})$. Thus their decomposition causes them to incur an extra factor of 2 in the exponent. They similarly incur a factor of 2 increase in exponent for the submodular width algorithm.

5.4 MapReduce and Parallel Processing

The GYM algorithm [3] uses GHDs to efficiently process joins in a MapReduce setting. GYM makes use of the GHD structure to parallelize different parts of the join. Given a GHD of depth d , and width w , with n attributes, GYM can perform a join in a MapReduce setting in $O(d + \log(n))$ rounds at a communication cost of $M^{-1}(\text{IN}^w + \text{OUT})^2$ where M is the memory per processor on the MapReduce cluster. Combining this with the degree-based MapReduce algorithm [13] gives us the following result:

► **Theorem 40.** *Given an optimal valid GHD (\mathcal{T}^*, χ) of depth d , and DBP-width $dbpw$, we can answer an AJAR query with Communication Cost equal to $O(M^{-1}(\text{IN}^{dbpw(\mathcal{T}, \mathcal{H})} + \text{OUT})^2)$ in $d + \log(n)$ MapReduce rounds, where n is the number of attributes and M is the available memory per processor.*

A GHD can have depth up to $O(n)$, in which case the algorithm can take a very large number of MapReduce rounds ($O(n)$). To address this, the GYM paper uses the ‘Log-GTA’ algorithm to reduce the depth of any given GHD to $\log(n)$ while at most tripling its width. This lets it process joins in $\log(n)$ MapReduce rounds at a cost of $M^{-1}(\text{IN}^{3w} + \text{OUT})^2$.

Log-GTA involves some shuffling of the attributes in the GHD bags, so naively applying it to a valid GHD could make the GHD invalid (see example 61 in the Appendix). But our decomposition lets us apply Log-GTA to the GHD of each characteristic hypergraph, and then stitch the short GHDs together. Our decomposition is recursive in nature; let d' be the maximum recursive depth of the decomposition for a given $Q_{\mathcal{H}, \alpha}$. Then the depth of the shortened GHD of each characteristic hypergraph is $O(\log(n))$, and so the depth of the valid GHD obtained by stitching them together is $O(d' \log(n))$. This gives us the result:

► **Theorem 41.** *If $dbpw$ is the DBP width of a AJAR query, we can answer that query with Communication Cost equal to $O(M^{-1}(\text{IN}^{3 \times dbpw(\mathcal{T}, \mathcal{H})} + \text{OUT})^2)$ in $d' \log(n)$ MapReduce rounds, where n is the number of attributes and M is the available memory per processor.*

d' can vary from $O(1)$ to $O(n)$ depending on the query. The star query from example 35 has $d' = 2$, which lets us process it in $\log(n)$ MapReduce rounds. Any query that only has a single type of aggregation will have $d' = 2$ as well. On the other hand, a query with one relation having n attributes, 1 output attribute, and alternating \sum and max aggregations, will have $d' = n$, and will be hard to parallelize.

Olteanu and Zavodny [6, 21] use valid GHDs to answer AJAR queries for the special case of a single type of aggregation. But they have no notion of a decomposition and attempting to shorten a valid GHD directly, without using a decomposition, may make it invalid. FAQ’s decomposition may be used to shorten GHDs similarly to ours, but leads to an increased width of $4faqw$ compared to our $3w$ (where $w \leq faqw$ is the width of our optimal valid GHD). This is again because of output addition, if the output attributes have a width of $faqw$, and the shortened GHDs of the characteristic hypergraphs have a width of $3faqw$, then the total width will be $4faqw$.

6 Product Aggregations

The primary application of queries with multiple aggregations is to establish bounds for the Quantified Conjunctive Query (QCQ) problem [15], and its counting variant, $\#QCQ$. We now

introduce a new type of aggregation, called product aggregation, that lets us efficiently handle *QCQ* queries. We define the AJAR problem for product aggregations, and then extend our algorithm from Section 3.3 to handle this new type of AJAR query. We then define a decomposition analogous to that in Section 5. A more detailed version of this section with additional motivation, examples, and proofs can be found in Appendix E.

6.1 AJAR queries with product aggregates

A *product aggregation* aggregates using the \otimes operator. Throughout the paper, we assumed that an absent tuple effectively has an annotation of 0. Taking this into account, we formally define the product aggregation. Let $B = F \setminus A$:

► **Definition 42.** $\sum_{(A, \otimes)} R_{AB} = \{(t_B, \lambda) : \forall t_A \in \mathcal{D}^A, t_B \circ t_A \in R_{AB} \text{ and } \lambda = \prod_{(t, \lambda_t) \in R_{AB}: \pi_B t = t_B} \lambda_t\}$

We can adjust the definition of aggregation orderings and AJAR queries to allow this new type of aggregation. *QCQ* queries can now be expressed as AJAR queries on the $(\{0, 1\}, \max, \cdot)$ semiring. We assume for this section that \otimes is idempotent, i.e. $a \otimes a = a$ for all a . We describe how to work with non-idempotent products in Appendix E.4.

6.2 Algorithms for product aggregates

For aggregation orderings that have product aggregations, the rules for determining when two orderings are equivalent are somewhat different; product aggregations can be performed *before* a join. We illustrate this with an example:

► **Example 43.** In the semiring $(\{0, 1\}, \max, \cdot)$, suppose we have two relations $R(A, B) = \{((0, 0), x), ((0, 1), y)\}$ and $S(B, C) = \{((0, 1), p), ((1, 1), q)\}$. Consider the AJAR query $\sum_{(B, \cdot)} R(A, B) \bowtie S(B, C)$. If we compute the join, we will get two tuples with the annotations $x \cdot p$ and $y \cdot q$, and then aggregating over B will produce a relation with the element $((0, 1), x \cdot p \cdot y \cdot q)$. However, note that $x \cdot p \cdot y \cdot q = (x \cdot y) \cdot (p \cdot q)$, implying that $\sum_{(B, \cdot)} R(A, B) \bowtie S(B, C) = (\sum_{(B, \cdot)} R(A, B)) \bowtie (\sum_{(B, \cdot)} S(B, C))$.

Now we describe our algorithm for solving AJAR queries when product aggregations are present. Our algorithm follows the same lines as the algorithm from Section 3.3. Recall that the algorithm consisted of searching for *equivalent orderings*, then searching for GHD *compatible* with an equivalent ordering, and running AggroGHDJoin on the GHD with the smallest fhw. For product aggregations, we need to modify our algorithm for testing equivalent orderings, and our definition of compatibility; we do these in turn.

Testing orderings for equivalence

We describe how we modify Algorithm 2 when product aggregations are present. Let $\text{PA}(\alpha)$ denote the set of product attributes in ordering α . We make two changes to Algorithm 2. (1) Instead of removing $V(-\alpha)$ and dividing \mathcal{H} into components, we remove $V(-\alpha) \cup \text{PA}(\alpha)$ and then divide \mathcal{H} into components C_1, C_2, \dots, C_m . Then for each C_i we define $C'_i = C_i \cup \bigcup_{e \in \mathcal{E}_{C_i}} (\text{PA}(\alpha) \cap e)$.⁷ That is C'_i has the attributes of C_i as well as the product attributes that are in the same hyperedge as some attribute in C_i . Then we recursively call the equivalence test on $(\alpha_{C'_i}, \beta_{C'_i})$ instead of on $(\alpha_{C_i}, \beta_{C_i})$. (2) When we are checking for a $i < j$ such that $\odot'_i \neq \odot'_j$ and there is a path in $\{b_i, b_{i+1}, \dots, b_{|\alpha|}\}$, we instead check for a path in

$$(\{b_i, b_{i+1}, \dots, b_{|\alpha|}\} \setminus \text{PA}(\alpha)) \cup \{b_i, b_j\}$$

⁷ Recall for any $V \subseteq \mathcal{V}$, \mathcal{E}_V is defined to be $\{E \in \mathcal{E} | E \cap V \neq \emptyset\}$.

That is, we look for a b_i that has a different operator than b_j , and has a path to b_j consisting only of b_i , b_j , and semiring attributes in $\{b_i, b_{i+1}, \dots, b_{|\alpha|}\}$. Appendix E gives the pseudo-code for the modified algorithm (Algorithm 7) and proves that it is sound and complete.

► **Lemma 44.** *The above Algorithm returns True if and only if $\alpha \equiv_{\mathcal{H}} \beta$.*

Compatible GHDs

Product aggregates not only change the set of equivalent orderings, but also the set of GHDs compatible with a given ordering. In fact, product aggregates allow us to break the rules of GHDs without causing incorrect behavior. We express this using a simple variant of GHDs, called *aggregating generalized hypertree decompositions* (AGHDs). Informally, AGHDs are GHDs that can violate the running intersection property for attributes that have a product aggregation. AGHDs are formally defined in Appendix E. We determine compatibility for AGHDs as follows: An AGHD is compatible with an ordering β if for every attribute pair a, b such that one of the $TOP(a)$ nodes is an ancestor of a $TOP(b)$ node, a precedes b in β .

We can now modify our algorithm from Section 3.3 to detect equivalent orderings using Algorithm 7, then search for compatible AGHDs, and run AggroGHDJoin over the compatible AGHD with the smallest fhw. Our runtime is given by the next theorem.

► **Theorem 45.** *Given a AJAR query $Q_{\mathcal{H}, \alpha}$ possibly involving idempotent product aggregates, let w^* be the smallest fhw for an AGHD compatible with an ordering equivalent to α . Then the runtime for our algorithm is $\tilde{O}(\text{IN}^{w^*} + \text{OUT})$.*

Decomposing AGHDs

We can apply the ideas from Section 5 to AJAR queries with product aggregates as well. We can define a notion of decomposable AGHDs for queries with product aggregates, and show the following results:

► **Theorem 46.** *All decomposable AGHDs are compatible with an ordering β such that $\beta \equiv_{\mathcal{H}} \alpha$.*

► **Theorem 47.** *For every valid AGHD (\mathcal{T}, χ) , there exists a decomposable (\mathcal{T}', χ') such that for all node-monotone functions γ , the γ -width of (\mathcal{T}', χ') is no larger than the γ -width of (\mathcal{T}, χ) .*

We can define characteristic hypergraphs similarly to how we did in Section 5 (see Appendix E for a formal definition). We have the following result:

► **Theorem 48.** *For an AJAR query $Q_{\mathcal{H}, \alpha}$ involving product aggregates, suppose $\mathcal{H}_0, \dots, \mathcal{H}_k$ are the characteristic hypergraphs $H(\mathcal{H}, \alpha)$. Then GHDs G_0, G_1, \dots, G_k of $\mathcal{H}_0, \dots, \mathcal{H}_k$ can be connected to form a decomposable AGHD G for $Q_{\mathcal{H}, \alpha}$. Conversely, any decomposable AGHD G of $Q_{\mathcal{H}, \alpha}$ can be partitioned into GHDs G_0, G_1, \dots, G_k of the characteristic hypergraphs $\mathcal{H}_0, \dots, \mathcal{H}_k$. Moreover, in both of these cases, γ -width(G) = $\max_i \gamma$ -width(G_i).*

These theorems let us apply all the optimizations from Section 5.2, 5.3, and 5.4 to AJAR queries with product aggregates.

Comparison to FAQ

The runtime of InsideOut on a query involving idempotent product aggregations is given by $\tilde{O}(\text{IN}^{faqw})$, where the faqw depends on the ordering, and the presence of product aggregations. Our algorithm for handling product aggregations recovers the runtime of FAQ. Formally,

► **Theorem 49.** *For any AJAR query involving idempotent product aggregations, $\text{IN}^{w^*} + \text{OUT} \leq 2 \cdot \text{IN}^{faqw}$.*

The proof is in Appendix B.1. By applying ideas from the FAQ paper to our setting, we can also recover the FAQ runtime on $\#QCQ$ (Appendix E.3). Algorithm 7 for detecting equivalence of orderings is both sound and complete; in contrast, FAQ’s equivalence testing algorithm is sound but not complete. Moreover, we have a width-preserving decomposition for queries with product aggregates. This allows us to get tighter runtime exponents in terms of submodular and DBP-widths (Theorems 38, 39) and efficient MapReduce Algorithms (Theorems 40, 41).

7 Conclusion

We investigate solutions to and the structure of AJAR queries: aggregate-join queries with multiple aggregators over annotated relations. We start by providing a very simple algorithm based on a variant of the standard GHDJoin algorithm that generates query plans by relying on a simple test of equivalence between aggregation orderings. This naive approach is sufficient to recover and surpass the runtime of state-of-the-art solutions. We proceed to investigate more interesting technical questions regarding the structure of AJAR queries. We first develop a partial ordering that fully characterizes equivalent orderings. We then develop a characterization of the corresponding valid GHDs, describing how they can be decomposed into ordinary, unrestricted GHDs. This reduction connects us to a trove of parallel work on GHDs. We finish by extending our work to handle product aggregations.

Acknowledgements

We thank Atri Rudra for invaluable insights and feedback developing our approach. CR gratefully acknowledges the support of the Defense Advanced Research Projects Agency (DARPA) XDATA Program under No. FA8750-12-2-0335 and DEFT Program under No. FA8750-13-2-0039, DARPAs MEMEX program under No. FA8750-14-2-0240, the National Science Foundation (NSF) under CAREER Award No. IIS-1353606, Award No. CCF-1356918 and EarthCube Award under No. ACI-1343760, the Office of Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, the Sloan Research Fellowship, the Moore Foundation Data Driven Investigator award, and gifts from American Family Insurance, Google, Lightspeed Ventures, and Toshiba.

References

- 1 Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *CoRR*, abs/1503.02368, 2015.
- 2 Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.
- 3 Foto N. Afrati, Manas Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.
- 4 Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, 2000.
- 5 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *FOCS ’08*, pages 739–748, Washington, DC, USA, 2008. IEEE Computer Society.
- 6 Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.
- 7 Hubie Chen and Víctor Dalmau. Decomposing quantified conjunctive (or disjunctive) formulas. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25–28, 2012*, pages 205–214, 2012.

- 8 Stephen Dolan. Fun with semirings: A functional pearl on the abuse of linear algebra. *SIGPLAN Not.*, 48(9):101–110, September 2013.
- 9 G. Gottlob, M. Grohe, M. Nysret, S. Marko, and F. Scarcello. Hypertree Decompositions: Structure, Algorithms, and Applications. In *WG*, 2005.
- 10 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *PODS 1999*, pages 21–32, 1999.
- 11 Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. *PODS '07*, pages 31–40, New York, NY, USA, 2007. ACM.
- 12 Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, August 2014.
- 13 Manas R. Joglekar and Christopher Ré. It’s all a matter of degree: Using degree information to optimize multiway joins. In *ICDT (to appear)*, 2016.
- 14 Kalev Kask, Rina Dechter, Javier Larrosa, and Avi Dechter. Unifying tree decompositions for reasoning in graphical models. *Artif. Intell.*, 166:165–193, 2005.
- 15 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. *CoRR*, abs/1504.04044, 2015.
- 16 Dániel Marx. Approximating fractional hypertree width. *ACM Trans. Algorithms*, 6, 2010.
- 17 Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing, STOC '10*, pages 735–744, New York, NY, USA, 2010. ACM.
- 18 Daniel Marx. Tractable structures for constraint satisfaction with truth tables. *Theory of Computing Systems*, 48, 2011.
- 19 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS 2012*, pages 37–48, 2012.
- 20 Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.
- 21 Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2, 2015.
- 22 Adam Perelman and Christopher Ré. Duncecap: Compiling worst-case optimal query plans. In *SIGMOD 2015*, pages 2075–2076, 2015.
- 23 Christopher Ré and Dan Suciu. The trichotomy of HAVING queries on a probabilistic database. *VLDB J.*, 18(5):1091–1116, 2009.
- 24 Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.
- 25 Charles A. Sutton and Andrew McCallum. An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4):267–373, 2012.
- 26 Susan Tu and Christopher Ré. Duncecap: Query plans using generalized hypertree decompositions. In *SIGMOD 2015*, pages 2077–2078, 2015.
- 27 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT 2014*, pages 96–106, 2014.
- 28 M. Yannakakis. Algorithms for Acyclic Database Schemes. In *VLDB*, 1981.

A Background

The AggroGHDJoin algorithm is a simple variant of some well-known join algorithms. We describe these algorithms next.

Algorithm 3 $\text{GHDJoin}(\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H}), (\mathcal{T}(\mathcal{V}_\mathcal{T}, \mathcal{E}_\mathcal{T}), \chi), \{R_F | F \in \mathcal{E}_\mathcal{H}\})$

Input: Query hypergraph \mathcal{H} , GHD (\mathcal{T}, χ) , Relations R_F for each $F \in \mathcal{E}_\mathcal{H}$

```

1:  $S_R \leftarrow \emptyset$ 
2: for all  $t \in \mathcal{V}_\mathcal{T}$  do
3:    $\mathcal{H}_t \leftarrow (\chi(t), \{\pi_{\chi(t)} F | F \in \mathcal{E}_\mathcal{H}\})$ 
4:    $S_R \leftarrow S_R \cup GJ(\mathcal{H}_t, \{\pi_{\chi(t)} R_F | F \in \mathcal{E}_\mathcal{H}\})$ 
5: end for
6: return  $Yannakakis(\mathcal{T}, S_R)$ 

```

GenericJoin

We first describe the AGM bound on the join output size developed by Atserias, Grohe, and Marx [5]. Given query hypergraph $\mathcal{H}_Q = (\mathcal{V}, \mathcal{E})$ and relations $\{R_F | F \in \mathcal{E}\}$, consider the following linear program:

$$\begin{aligned} \text{Minimize } & \sum_{F \in \mathcal{E}} x_F \log_{\text{IN}}(|R_F|) \\ \forall v \in \mathcal{V} : & \sum_{F: v \in F} x_F \geq 1 \\ \forall F \in \mathcal{E} : & x_F \geq 0 \end{aligned}$$

Any feasible solution \vec{x} is a *fractional edge cover*. Suppose ρ^* is the optimal objective. Then the *AGM bound* on the worst-case output size of join $\bowtie_{F \in \mathcal{E}} R_F$ is given by $\text{IN}^{\rho^*} = \prod_{F \in \mathcal{E}} |R_F|^{x_F^*}$. We will use $\text{IN}^{\text{AGM}(Q)}$ to denote the AGM bound on a query Q . The GenericJoin (GJ) algorithm [20] computes a join in time $\tilde{O}(\text{IN}^{\text{AGM}(Q)})$ for any join query. GJ will be used as a subroutine in a later algorithm, where $GJ(\mathcal{H}, \{R_F | F \in \mathcal{E}_\mathcal{H}\})$ denotes a call to GenericJoin with one input relation R_F per hyperedge F in hypergraph \mathcal{H} .

Yannakakis

Yannakakis' algorithm [28] operates on α -acyclic queries. There are several different equivalent definitions of α -acyclicity; we provide the definition that builds a tree out of the relations as it most naturally relates to generalized hypertree decompositions.

► **Definition 50.** Given a hypergraph $\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$, a *join tree* over \mathcal{H} is a tree $\mathcal{T} = (\mathcal{V}_\mathcal{T}, \mathcal{E}_\mathcal{T})$ with $\mathcal{V}_\mathcal{T} = \mathcal{E}_\mathcal{H}$ such that for every attribute $A \in \mathcal{V}_\mathcal{H}$, the set $\{v \in \mathcal{V}_\mathcal{T} | A \in v\}$ forms a connected subtree in \mathcal{T} .

A hypergraph \mathcal{H} is α -acyclic if there exists a join tree over \mathcal{H} [2, 28]. We can use the classic GYO algorithm to produce a join tree [2, ch.6]. The Yannakakis algorithm takes a join tree as input. Its pseudo-code is given in Section 3.1.

► **Theorem 51.** *Algorithm 1 runs in $O(\text{IN} + \text{OUT})$ where IN and OUT are the sizes of the input and output, respectively.*

To leverage the speed of Yannakakis for cyclic queries, we look to GHDs [9, 12]. The intuition behind a GHD is to group the attributes into bags (as specified by the function χ) such that we can build a *join tree* over these bags. This allows us to run *GJ* within each bag and then Yannakakis on the join tree. The resulting algorithm is *GHDJoin* whose pseudo-code is given in Algorithm 3. The runtime of *GHDJoin* is given by $\tilde{O}(\text{IN}^{\text{fhw}(\mathcal{T}, \mathcal{H})} + \text{OUT})$

► **Theorem 52.** *Algorithm 3 runs in $\tilde{O}(\text{IN}^{\text{fhw}(\mathcal{T}, \mathcal{H})} + \text{OUT})$.*

We can make some straightforward modifications to the above join algorithms to perform aggregations. The traditional Yannakakis and GHDJoin algorithms perform the join in a bottom up fashion, after a semijoin phase to ensure that there are no dangling tuples. The modified algorithms above handle aggregations using the same intuition as in traditional query plans: “push down” aggregations as far as possible. Since each attribute must occur in a connected subtree of the GHD, we can push its aggregation down to the root of this connected subtree, which is the TOP node of the attribute. There is a standard modification to Yannakakis for project-join queries that projects away attributes at their TOP node [28]. Instead of projecting, we perform aggregation.

We provide the pseudo-code of AggroYannakakis, which is a simple variant of the well-known Yannakakis [28] algorithm, in Algorithm 4. Algorithm 5 gives the pseudo-code of AggroGHDJoin, which is a variant of GHDJoin that calls AggroYannakakis instead of Yannakakis. AggroGHDJoin also does some extra work to ensure we pass each annotation to GJ only once. The π^1 operator in AggroGHDJoin denotes a projection that projects tuples while replacing the annotation by 1, to ensure that the same annotation isn’t counted more than once.

Algorithm 4 AggroYannakakis($\mathcal{T} = (\mathcal{V}, \mathcal{E})$, α , $\{R_F | F \in \mathcal{V}\}$)

Input: Join tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, Aggregation order α , Relations R_F for each $F \in \mathcal{V}$

```

for all  $F \in \mathcal{V}$  in some bottom-up order do                                 $\triangleright$  Semi-join reduction up
   $P \leftarrow$  parent of  $F$ 
   $R_P \leftarrow R_P \bowtie R_F$ 
end for
for all  $F \in \mathcal{V}$  in some top-down order do                                 $\triangleright$  Semi-join reduction down
   $P \leftarrow$  parent of  $F$ 
   $R_F \leftarrow R_F \bowtie R_P$ 
end for
while  $F \in \mathcal{V}$  in some bottom-up order do                                 $\triangleright$  Aggregation
   $\beta \leftarrow \alpha \cap \{a \in \mathcal{V} | TOP_{\mathcal{T}}(a) = F\}$ 
   $R' \leftarrow \Sigma_{\beta} R_F$ 
  if  $F$  is not the root then
     $P \leftarrow$  parent of  $F$ 
     $R_P \leftarrow R_P \bowtie R'$                                           $\triangleright$  Compute the join
  end if
end while
return  $R_R$  for the root  $R$ 

```

Algorithm 5 AggroGHDJoin($\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, $(\mathcal{T}(\mathcal{V}_{\mathcal{T}}, \mathcal{E}_{\mathcal{T}}), \chi)$, $\{R_F | F \in \mathcal{E}_{\mathcal{H}}\}$)

Input: Query hypergraph \mathcal{H} , GHD (\mathcal{T}, χ) , Relations R_F for each $F \in \mathcal{E}_{\mathcal{H}}$

```

 $S_R \leftarrow \emptyset$ 
for all  $t \in \mathcal{V}_{\mathcal{T}}$  do
   $\mathcal{H}_t \leftarrow (\chi(t), \{\pi_{\chi(t)} F | F \in \mathcal{E}_{\mathcal{H}}\})$ 
   $I \leftarrow \{R_F | F \subseteq \chi(t), \exists a \in F : TOP_{\mathcal{T}}(a) = t\} \cup \{\pi_{\chi(t)}^1 R_F | F \not\subseteq \chi(t) \text{ or } \forall a \in F : TOP_{\mathcal{T}}(a) \neq t\}$ 
   $S_R \leftarrow S_R \cup GJ(\mathcal{H}_t, I)$ 
end for
return AggroYannakakis( $\mathcal{T}, S_R$ )

```

In the classic analysis of Yannakakis, the runtime of the semi-join portion is bounded by $O(\text{IN})$ and the bottom-up join is bounded by $O(\text{OUT})$. In AggroYannakakis, the analysis of the semi-join portion is unchanged, but the aggregation reduces the size of the output, thereby making the OUT bound harder to achieve. In particular, during the bottom-up join, we may compute an intermediate relation whose attributes are not a subset of the output attributes, meaning that its size may not be bounded by OUT. These potentially large intermediate relations are the underlying cause for the traditional $\text{IN} \cdot \text{OUT}$ runtime.

However, using intuition discovered in reference [6], if we require the output attributes to appear above non-output attributes, we can preserve the $\text{IN} + \text{OUT}$ runtime.

► **Theorem 53.** *Suppose we have a GHD such that for any output attribute A and non-output attribute B , $\text{TOP}_{\mathcal{T}}(B)$ is not an ancestor of $\text{TOP}_{\mathcal{T}}(A)$. AggroGHDJoin runs in $\tilde{O}(\text{IN}^{\text{fhw}(\mathcal{T}, \mathcal{H})} + \text{OUT})$ given this GHD.*

Proof. GJ on each bag still runs in $\tilde{O}(\text{IN}^{\text{fhw}(\mathcal{T}, \mathcal{H})})$. We need to prove the Yannakakis portion runs in $O(\text{IN} + \text{OUT})$ after running GJ .

The semijoin portion runs in $O(\text{IN})$ as in the original Yannakakis algorithm. In the join phase, we have two types of joins. In the first type, $F \setminus \beta \subseteq P$. This implies the join output is a subset of R_P (with different annotations). So the total runtime of this type of join is $O(\text{IN})$. For the second type, $F \setminus \beta \not\subseteq P$. This means some attribute in $(F \setminus \beta) \setminus P$ must be an output attribute, and all attributes in P must be output attributes as well (as their TOP value is an ancestor of F). So the result of our join must be a subset of the output table; the total runtime of this type of join is $O(\text{OUT})$. Thus the total runtime of the algorithm is $O(\text{IN} + \text{OUT})$. ◀

Note that while AggroGHDJoin runs in $\tilde{O}(\text{IN}^{\text{fhw}(\mathcal{T}, \mathcal{H})} + \text{OUT})$ time on the GHDs above, it may not necessarily produce the right output unless the GHD satisfies additional conditions, to ensure that aggregations can be done in the proper order. In particular, recall our definition a GHD (\mathcal{T}, χ) is compatible with α if for all attribute pairs A, B , $\text{TOP}_{\mathcal{T}}(A)$ being an ancestor of $\text{TOP}_{\mathcal{T}}(B)$ implies that either A is an output variable or A occurs before B in α .

► **Theorem 54.** *If a GHD (\mathcal{T}, χ) is compatible with α , then AggroGHDJoin given (\mathcal{T}, χ) correctly computes $Q_{\mathcal{H}, \alpha}$.*

Proof. We first show that AggroYannakakis works as expected. We note that the semi-join reduction does not change the output; it only quickens the process. We only consider the bottom-up join. For each node t in the join tree, let $R(t)$ be the relation associated with that node before this loop (i.e. after the semi-join portion). Let $R'(t)$ be the final relation associated with node t when we are processing node t (i.e. after the bottom up join with t 's descendants is done, and after the aggregation in t). Let \mathcal{T}_t be the subtree that includes t and all of its descendants. Let $s(t)$ be the attributes aggregated at node t , i.e. $\alpha \cap \{a \in \mathcal{V} | \text{TOP}_{\mathcal{T}}(a) = t\}$, and let $s(\mathcal{T}_t) = \cup_{t \in \mathcal{T}_t} s(t)$. For each non-leaf node t , let $c(t)$ be the set of t 's children.

For each node t , we claim $R'(t) = \sum_{\alpha_{s(\mathcal{T}_t)}} \bowtie_{t' \in \mathcal{T}_t} R(t')$. Proof by induction on the tree. For each leaf l , $R'(l) = \sum_{\alpha_{s(l)}} R(l)$ by definition.

For a non-leaf node t ,

$$\begin{aligned} R'(t) &= \sum_{\alpha_{s(t)}} R(t) \bowtie (\bowtie_{t_c \in c(t)} R'(t_c)) \\ &= \sum_{\alpha_{s(t)}} R(t) \bowtie \left(\bowtie_{t_c \in c(t)} \sum_{\alpha_{s(\mathcal{T}_{t_c})}} \bowtie_{t' \in \mathcal{T}_{t_c}} R(t') \right) \\ &= \sum_{\alpha_{s(\mathcal{T}_t)}} \bowtie_{t' \in \mathcal{T}_t} R(t') \end{aligned}$$

The second step is due to the inductive hypothesis. The final step is simply “pulling out” the aggregations from the sub-orderings one at a time; we can arbitrarily interleave the aggregation orders $\alpha_{s(\mathcal{T}_t)}$. We can simply interleave them to match $\alpha_{\cup_{t_c \in c(t)} s(\mathcal{T}_t)}$. Since the original GHD is compatible with α , we know the aggregations $\alpha_{s(t)}$ precede $\alpha_{s(\mathcal{T}_t)}$ in α , implying that $\sum_{\alpha_{s(\mathcal{T}_t)}} \sum_{\alpha_{\cup_{t_c \in c(t)} s(\mathcal{T}_t)}} = \sum_{\alpha_{s(t)}}$. Our output is $R'(t_r)$ where t_r is the root node, which is $\sum_{\alpha} \bowtie_{t \in \mathcal{T}} R(t)$ as desired.

Since AggroYannakakis works as expected, we simply need to ensure that the bags are computed appropriately. Note the GHD ensures for every relation R_F , there is a node t such that $F \subseteq \chi(t)$. This means that no tuple is lost; computing AggroYannakakis on the bags will compute the correct tuples. To ensure it computes the correct annotations, we need to ensure every annotation appears in the bags at most once; our algorithm places the annotation of a relation R_F in the top-most node that contains all of the attributes R_F . \blacktriangleleft

Product Aggregations: When product aggregations are present in an AJAR query $Q_{\mathcal{H}, \alpha}$, we have a notion of product partition hypergraphs, AGHDs over product partition hypergraphs, and a corresponding notion of AGHDs compatible with an ordering. We now prove theorem 45 that extends theorems 53 and 54 to the case where product aggregations are present.

A product partition partition $P = (\mathcal{V}_P, \mathcal{E}_P)$ essentially creates multiple renamed copies of each product attribute a ($a_1, a_2, \dots, a_{|P_a|}$), and assigns one of the renamed copies to each relation containing a . An AGHD is essentially a GHD over P . Given P , and $a \in \mathcal{V}_H$, let $P(a)$ equal $\{a\}$ if a is not a product attribute, and $\{a_1, \dots, a_{|P_a|}\}$ otherwise. Given $a' \in \mathcal{V}_P$, let $P^{-1}(a')$ equal a such that $a' \in P(a)$. Given an edge $F \in \mathcal{E}_P$, let $P^{-1}(F)$ denote the edge $\{P^{-1}(a') \mid a' \in F\}$. We define a modified ordering α^P over \mathcal{V}_P that takes α and replaces each occurrence of (a, \otimes) with $(a_1, \otimes), (a_2, \otimes), \dots, (a_{|P_a|}, \otimes)$ for each product attribute a . For any $F \in \mathcal{E}_P$, we define the relation R_F to be same as the $R_{P^{-1}(F)}$ (but with the attribute name changed). This gives us the modified AJAR query $Q_{P, \alpha^P}^P = \sum_{\alpha^P} \bowtie_{F \in \mathcal{E}_P} R_F$. Then we have,

► **Lemma 55.** *Suppose $R'(A, C_1)$ is a copy of $R(A, C)$ with C renamed to C_1 , and $S'(B, C_2)$ is a copy of $S(B, C)$ with C renamed to C_2 . Then*

$$\sum_{(C_1, \otimes)} \sum_{(C_2, \otimes)} R'(A, C_1) \bowtie S'(B, C_2) = \sum_{(C, \otimes)} R(A, C) \bowtie S(B, C)$$

Proof. Suppose the annotations for the C values in R are n_1, n_2, \dots, n_k and in S are m_1, m_2, \dots, m_k (assume all annotations are present i.e. absent tuples have a zero-annotation). Then the RHS is $\otimes_{i=1}^k n_i m_i$. The LHS will have $\otimes_{i=1}^k n_i \otimes_{j=1}^k m_j$. The RHS is equal to the LHS because of idempotence of \otimes . Note that if \otimes wasn’t idempotent, the LHS would have the m_j terms multiplied k times while the RHS has them once. \blacktriangleleft

► **Lemma 56.** *For each database instance I , $Q_{\mathcal{H}, \alpha}(I) = Q_{P, \alpha^P}^P(I)$.*

This lemma can be proved by repeated application of Lemma 55.

Now we can easily prove Theorem 45. Suppose we have a AGHD $D = (\mathcal{T}, \chi, P)$ which is compatible with an ordering α . Then the GHD (\mathcal{T}, χ) over hypergraph P , is compatible with α^P . Running AggroGHDJoin over this GHD, with ordering α^P correctly computes $Q_{P, \alpha^P}^P(I)$, due to theorem 54. And by Lemma 56, this also equals $Q_{\mathcal{H}, \alpha}(I)$, which is the output we want. Also, since the AGHD is compatible with α , the GHD must satisfy the condition of Theorem 53, and hence AggroGHDJoin runs on it in time $\tilde{O}(\text{IN}^{fhw} + \text{OUT})$.

B Comparison with Related Work

B.1 Section 3

In Section 3, we define a simple approach to solving AJAR queries, and we claim in Theorem 12 that our runtime guarantee of $\tilde{O}(IN^{w^*} + \text{OUT}) \leq \tilde{O}(IN^{faqw})$. We note that the $faqw$ exponent is actually the optimum value of $faqw(\sigma)$ over the equivalent orderings σ they consider (we discuss the space of orderings they consider in the next subsection). Our approach will recognize σ as being equivalent, and will search for the best compatible GHD for σ . We will show that there exists a compatible AGHD (\mathcal{T}, χ, P) for every equivalent ordering σ such that $fhw(\mathcal{T}, \mathcal{H}) = faqw(\sigma)$ (as Example 58 shows, the compatible AGHD \mathcal{T} we construct may not be the optimal compatible GHD).

We start by briefly summarizing FAQ's algorithm, with the pseudo-code (written in the notation of this paper) given in Algorithm 6. Let σ be the ordering used for aggregation. Let n denote the total number of attributes $|\mathcal{V}_{\mathcal{H}}|$ and f denote the number of output attributes (thus $|\sigma| = n - f$). For notational convenience, we will be using $\sigma[i]$ to denote both the attribute and the operator that make up the i^{th} operator-attribute pair in the ordering.

Algorithm 6 `InsideOut`($\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, σ , $\{R_F | F \in \mathcal{E}_{\mathcal{H}}\}$)

Input: Hypergraph $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, Aggregation ordering σ , Relations R_F for each $F \in \mathcal{E}_{\mathcal{H}}$

```

 $E_n \leftarrow \{R_F | F \in \mathcal{E}_{\mathcal{H}}\}$ 
for  $(k = n; k > f; k --)$  do
   $\delta(k) \leftarrow \{R_F \in E_k | \sigma[k - f] \in F\}$ 
  if  $\sigma[k - f]$  is not a product aggregation then
     $U_k \leftarrow \bowtie_{R \in \delta(k)} R$ 
     $E_{k-1} \leftarrow (E_k \setminus \delta(k)) \cup \{\sum_{\sigma[k-f]} U_k\}$ 
  else
     $E_{k-1} \leftarrow (E_k \setminus \delta(k)) \cup \{\sum_{\sigma[k-f]} R | R \in \delta(k)\}$ 
  end if
end for
return  $\bowtie_{R \in E_f} R$ 

```

FAQ relies on a worst-case optimal algorithm to compute each of the joins, implying that in the $\tilde{O}(IN^{faqw})$ runtime guarantee, $faqw$ is defined as the maximum AGM bound placed on each of the computed joins. Define $p_H^* : 2^{\mathcal{V}_{\mathcal{H}}} \rightarrow \mathcal{R}$ to be a function that maps a subset of the attributes to the AGM bound on the subset (i.e. the optimal value of the canonical linear program). Then $faqw = \max(\max_k p_H^*(U_k), p_H^*(V(-\sigma)))$ [15].

We will build up the compatible AGHD (\mathcal{T}, χ, P) in rounds corresponding to each of the k values of *InsideOut*. We first describe how to construct (\mathcal{T}, χ) , and later describe how to obtain P . At the start of round corresponding to a particular k , we will have a forest of AGHDs, each of which will have a root mapped (by χ) to the attribute sets of E_k , and at the end of each round, the forest's roots will be mapped to the relations of E_{k-1} .

For an attribute set F , let $t(F)$ represent the node such that $\chi(t(F)) = F$. We start by creating the $|\mathcal{E}_{\mathcal{H}}|$ nodes $\{t(F) | F \in E_n\}$, which are simply nodes mapped to the input relations. Then for each k from n to $f + 1$, let T represent the set of nodes $\{t(F) | F \in \delta(n)\}$; these are the nodes that will be processed (i.e. the nodes for whom we will create parents). If $\sigma[k - f]$ is not a product aggregation, we create a node $t(U_k)$ and set $\text{parent}(t) = t(U_k)$ for all $t \in T$. We then create a node $t(U_k \setminus \{\sigma[k - f]\})$ and set it to be $\text{parent}(t(U_k))$. Note that this process has transformed the set of the forest's roots by removing T and adding $t(U_k \setminus \{\sigma[k - f]\})$, mirroring the transformation between E_k and E_{k-1} . If $\sigma[k - f]$ is a product aggregation, then for each

$F \in \delta(n)$, we create a node $t(F \setminus \{\sigma[k-f]\})$ and set it to be $\text{parent}(t(F))$; in this case as well the set of the forest's roots match E_{k-1} .

At the end of this process, we will have a forest of AGHDs whose roots map to the relations in E_f . To conclude our construction, we simply construct the node $t(V(-\sigma))$ and set it to be $\text{parent}(t(F))$ for all $F \in E_f$. If there are no product aggregations, then (\mathcal{T}, χ) forms a GHD.

(\mathcal{T}, χ) satisfies the running intersection property for all non-product attributes, but a product attribute a can be present in multiple disconnected parts of \mathcal{T} . We now describe a product partition P such that (\mathcal{T}, χ, P) forms an AGHD for the ordering σ . Let P_a denote the number of distinct connected components of \mathcal{T} in which a is present. Then we create P_a copies of a ($a_1, a_2, \dots, a_{|P_a|}$), and assign a copy to each component in some order. For each $F \in \mathcal{E}_H$ that contains a , if the component that $t(F)$ belongs to is assigned a_i , then P assigns a_i to F . Then (\mathcal{T}, χ, P) is an AGHD for σ .

The AGHD (\mathcal{T}, χ, P) as described is trivially compatible with σ since we construct $\text{parent}(TOP_{\mathcal{T}}(\sigma[k-f]))$ explicitly in round k ; this ensures that $TOP_{\mathcal{T}}(\sigma[i])$ cannot be an ancestor of $TOP_{\mathcal{T}}(\sigma[j])$ if $i > j$.

► **Lemma 57.** *Define $p_{\mathcal{H}}^*$ to be a function that maps a set of attributes to the AGM bound on the set (the optimal value of the canonical linear program). The AGHD (\mathcal{T}, χ, P) constructed as described satisfies*

$$fhw(\mathcal{T}, \mathcal{H}) = \max(p_{\mathcal{H}}^*(V(-\alpha)), \max_k p_{\mathcal{H}}^*(U_k)) = faqw(\sigma).$$

Proof. The nodes in our tree that do not map to $V(-\alpha)$ or the U_k either map to an input relation or to a relation created by aggregating an attribute from a single child node. In the former case, $p_{\mathcal{H}}^*$ would evaluate to 1, so we can ignore them in our maximum. In the latter case, the attributes are a strict subset of its child's attributes, implying we can ignore them too. As such, the fractional hypertree width is simply the maximum fractional cover over $V(-\alpha)$ and the U_k . This shows the first part of the equality.

The second part of the equality is the definition of $faqw$ [15]. ◀

Theorem 12, as well as its analogue for product aggregations, follow as a simple corollary. We now show an example where the runtime of InsideOut is much worse than the runtime of our Algorithm, primarily due to the fact that it is not output-sensitive.

► **Example 58.** Let n be an even number, and consider an AJAR query $Q_{\mathcal{H}, \alpha}$ where $\mathcal{H} = (\{A_i \mid 1 \leq i \leq n\}, \{\{A_i, A_{i+1}\} \mid 1 \leq i \leq n-1\} \cup \{\{A_n, A_1\}\})$, and α is empty (i.e. the query is just a join). Also let each attribute take values $1, 2, 3, \dots, 2 \times \lfloor \sqrt{N} \rfloor$. Suppose each relation $\{A_i, A_{i+1}\}$ for $1 \leq i \leq n-1$ connects values of the same parity, while relation $\{A_n, A_1\}$ connects values of opposite parities. Thus each relation has size N , and $\text{IN} = O(N)$ (n is a constant), and the join output is empty. There is a GHD with bags $\{A_1, A_2, A_3\}, \{A_1, A_3, A_4\}, \dots, \{A_1, A_{n-1}, A_n\}$ that is compatible with the empty ordering. The fhw of this GHD is 2, so we have $w^* = 2$. Thus the runtime of our algorithm will be $\tilde{O}(\text{IN}^2)$. InsideOut will compute an intermediate output consisting of the join of $n-1$ of the relations, which has size $N^{(n-1)/2}$, so InsideOut's runtime will be at least $\tilde{O}(\text{IN}^{(n-1)/2})$.

FAQ does discuss, at a very high-level and without proofs, changes to InsideOut that will allow their runtime to be output-sensitive [15, Section 10.2]. Their most general and useful change involves building a GHD for the output variables and running a message passing algorithm between the bags, which exactly describes GHDJoin. Implementing this change would make InsideOut completely equivalent to AggroGHDJoin. We note that the FAQ paper frames these changes as decisions in how to *represent* the output, whereas we present the optimization in an algorithmic context, independent of any other storage optimizations.

B.2 Section 4

In Section 4, we define a partial order $<_{\mathcal{H},\alpha}$ that exactly characterizes the constraints an aggregation ordering must satisfy to be equivalent to a given ordering α . Our partial ordering is complete, which is a result that FAQ cannot match. Much like our approach, FAQ actually defines their own partial ordering, which we denote $<_{FAQ}$, and their work only considers orderings that are linear extensions of $<_{FAQ}$. However, we will show an example where $<_{FAQ}$ has unnecessary constraints:

► **Example 59.** Consider the AJAR query given by $\sum_A \max_B \sum_C R(A, B)S(A, C)$. By our characterization, $A <_{\mathcal{H},\alpha} B$ is the only constraint, giving rise to 3 different valid orderings. The FAQ characterization, however, has two constraints: $A <_{FAQ} B$ and $C <_{FAQ} B$, which only allows for 2 different valid orderings. Note that FAQ constraints *preclude the original ordering* ABC.

B.3 Section 5

In Section 5, we define a decomposition that relates the width of a valid GHD to the widths of a series of ordinary GHDs. Variable orderings (as used by FAQ) are not as readily suited as GHDs are for decompositions. FAQ does derive their own version of a decomposition, but the difficulties that arise when using variable orderings are exemplified in the way FAQ switches between GHDs and variable orderings in their proofs [15]. In addition, the FAQ decomposition is demonstrably weaker than ours; their decomposition incurs some overhead costs when combining the sub-orderings to build the overall ordering, precluding a result like Corollary 32 that provides the groundwork for the variety of extensions we provide. To exemplify the gap in the two decompositions, we inspect a specific AJAR query:

► **Example 60.** Consider the query $\sum_B \sum_C \sum_D R(A, B)S(B, C)T(C, D)U(D, A)$. Suppose $|A| = \sqrt{N}$, $|B| = 2 = |D|$, $|C| = N$, and all of the pairwise relations are constructed as complete cross products of the attributes' values. Our decomposition will result in the chain GHD $A - ABD - BCD$, while the FAQ decomposition will result in the GHD $A - ABC - ACD$. The runtimes of both FAQ and GHDJoin using the former GHD is $\tilde{O}(N)$, whereas the runtimes using the latter GHD are $\tilde{O}(N^{3/2})$. As such, the FAQ decomposition will perform asymptotically worse than our decomposition.

More generally, consider a query Q_n with relations $R_i(A_i, B)$, $S_i(B, C_i)$, $T_i(C_i, D)$, $U_i(D, A_i)$ for $1 \leq i \leq n$. Like before, all $|A_i|$'s are \sqrt{N} , all $|C_i|$'s are N , and $|B| = |D| = 2$. And the aggregation ordering only has the + operator, on B , D and all C_i 's. Our decomposition gives the chain $A_1 \dots A_n - A_1 \dots A_n BD - C_1 \dots C_n BD$. This results in a runtime of $\tilde{O}(N^n)$. FAQ's decomposition gives $A_1 \dots A_n BC_1 \dots C_n - A_1 \dots A_n DC_1 \dots C_n$. This decomposition, and its corresponding ordering, give a runtime of $\tilde{O}(N^{3n/2})$. Thus the difference between runtime exponents caused by FAQ's decomposition and our decomposition can be arbitrarily high.

B.4 Section 5.4

► **Example 61.** Suppose we a AJAR query $Q_{\mathcal{H},\alpha}$ with

$$\mathcal{H} = (\{A, B, C, D, E, F\}, \{\{A, B\}, \{B, C\}, \{B, D, E\}, \{D, F\}\})$$

and $\alpha = ((D, \sum), (E, \sum), (F, \sum))$. We start with the width-1 valid GHD (\mathcal{T}, χ) with $\mathcal{V} - \mathcal{T} = \{v_1, v_2, v_3, v_4\}$ and

$$\mathcal{E}_{\mathcal{T}} = \{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}$$

such that v_1 is the root, and $\chi(v_1) = \{A, B\}$, $\chi(v_2) = \{B, C\}$, $\chi(v_3) = \{B, D, E\}$, $\chi(v_4) = \{D, F\}$.

Applying Log-GTA gives us a shorter GHD (\mathcal{T}', χ') with $\mathcal{V}'_{\mathcal{T}} = \{u, v_1, v_2, v_3, v_4\}$,

$$\mathcal{E}'_{\mathcal{T}} = \{\{v_1, u\}, \{u, v_2\}, \{u, v_3\}, \{u, v_4\}\}$$

with v_1 as the root. $\chi'(u) = \{B, D\}$ and $\chi'(v_i) = \chi(v_i)$ for all i . Now $TOP(D) = u$ which is an ancestor of $TOP(C) = v_2$, despite C being an output attribute and D not being an output attribute. This means GHD (\mathcal{T}', χ') is invalid, showing that applying Log-GTA to a valid GHD may make it invalid.

As the above example shows, we cannot directly apply Log-GTA to a valid GHD to get a shorter valid GHD.

C Characterizing Equivalent Orderings: Proofs

We now formally present our partial order $<_{\mathcal{H}, \alpha}$ that characterizes the interaction of the two forms of commuting. As we said in Section 4, we have two relations PREC and DNC, that are mutually recursive. We initialize the constraints to a base case and iteratively update them till we reach a fixed point. We now formalize this. We use binary operator $<_{\mathcal{H}, \alpha}^i$ to denote the constraint PREC after i iterations, and operator $\sim_{\mathcal{H}, \alpha}^i$ to denote DNC after i iterations, with one difference; both operators behave slightly differently for output attributes. To readily incorporate output attributes into the constraints, we define an augmented aggregation ordering below:

► **Definition 62.** For any aggregation ordering α , let F be the set of output variables. Then define $\alpha^O = \alpha_1^O, \alpha_2^O, \dots, \alpha_n^O$ to be a sequence such that $\alpha_i^O = (F_i, \text{NULL})$ for $1 \leq i \leq |F|$ and $\alpha_i^O = \alpha_{i+|F|}$ for $|F| + 1 \leq i \leq n$.

Note that n is defined to be the number of attributes in the query. Now we can formally define $<_{\mathcal{H}, \alpha}^i$ and $\sim_{\mathcal{H}, \alpha}^i$. Both of these binary operators operate over attribute-operator pairs, but since each attribute occurs at most once in an ordering, we can equivalently think of them as operating over attributes. We use these two interchangeably e.g. $A <_{\mathcal{H}, \alpha} B$ denotes the same thing as $(A, \oplus) <_{\mathcal{H}, \alpha} (B, \oplus')$.

► **Definition 63.** For a given query $Q_{\mathcal{H}, \alpha}$ with $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, we define relations $\sim_{\mathcal{H}, \alpha}^i$ and partial orders $<_{\mathcal{H}, \alpha}^i$ over attribute-operator pairs in α^O . For any $A, B \in \mathcal{V}$, suppose $(A, \oplus), (B, \oplus') \in \alpha^O$. Then, for $i = 0$, $(A, \oplus) \sim_{\mathcal{H}, \alpha}^0 (B, \oplus')$ if and only if one of the following is true:

■ $\oplus \neq \oplus'$ and $\exists E \in \mathcal{E} : A, B \in E$. (0.1)

■ $\oplus \neq \oplus'$ and either $\oplus = \text{NULL}$ or $\oplus' = \text{NULL}$ (0.2)

For $i > 0$, $(A, \oplus) \sim_{\mathcal{H}, \alpha}^i (B, \oplus')$ if and only if $(A, \oplus) \not\sim_{\mathcal{H}, \alpha}^j (B, \oplus')$ for all $j < i$ and one of the following is true:

■ $\oplus \neq \oplus'$ and $\exists E \in \mathcal{E}, (C, \oplus'') \in \alpha^O : B, C \in E, (A, \oplus) <_{\mathcal{H}, \alpha}^{i-1} (C, \oplus'')$ (i.1)

■ $\exists (C, \oplus'') \in \alpha^O$ and $j, k < i : (A, \oplus) <_{\mathcal{H}, \alpha}^j (C, \oplus'') <_{\mathcal{H}, \alpha}^k (B, \oplus')$ (i.2)

For any $i \geq 0$, $(A, \oplus) <_{\mathcal{H}, \alpha}^i (B, \oplus')$ if and only if $(A, \oplus) \sim_{\mathcal{H}, \alpha}^i (B, \oplus')$ and (A, \oplus) precedes (B, \oplus') in α^O .

Finally, $(A, \oplus) \sim_{\mathcal{H}, \alpha} (B, \oplus')$ if and only if $(A, \oplus) \sim_{\mathcal{H}, \alpha}^i (B, \oplus')$ for some $i \geq 0$. Similarly, $(A, \oplus) <_{\mathcal{H}, \alpha} (B, \oplus')$ if and only if $(A, \oplus) <_{\mathcal{H}, \alpha}^i (B, \oplus')$ for some $i \geq 0$.

The core of our definition is the four labeled conditions for \sim . The condition 0.1 represents the simplest structure that violates both conditions of Theorem 9; it represents our base case. Condition 0.2 simply ensures the output attributes precede non-output attributes. Our condition *i.1* extends the structure from 0.1 beyond single relations. If $A < C$ and C appears in a relation with B , we can guarantee that A and B cannot be separated in the way the second condition of Theorem 9 requires, and if $\oplus \neq \oplus'$, the first condition is violated as well. Condition *i.2* simply ensures that transitivity interacts properly with condition *i.1*.

We now prove the two lemmas stated in Section 4, followed by proving soundness and completeness of $<_{\mathcal{H},\alpha}$.

► **Lemma 64** (Copy of Lemma 23). *Suppose we are given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and an aggregation ordering α . Fix two arbitrary attributes $A, B \in \mathcal{V}$ such that $(A, \oplus), (B, \oplus') \in \alpha^O$ for differing operators $\oplus \neq \oplus'$. Then, for any path P in \mathcal{H} between A and B , there must exist some attribute in the path $C \in P$ such that $C <_{\mathcal{H},\alpha} A$ or $C <_{\mathcal{H},\alpha} B$.*

Proof. We use induction on the length of path P .

Base Case: Let $|P| = 2$. This implies that there exists some edge $E \in \mathcal{E}$ such that $A, B \in E$. Thus $A \sim_{\mathcal{H},\alpha}^0 B$. Then, by definition, either $A <_{\mathcal{H},\alpha} B$ or $B <_{\mathcal{H},\alpha} A$ depending on which attribute appears first in α .

Induction: Suppose $|P| = N > 2$ and assume the lemma is true for paths of length $< N$. We call this assumption the *outer inductive hypothesis*, for reasons that will become apparent later. Path P can be rewritten as $P = AP'B$ where P' is a path of length at least 1. Let C be the node in P' that appears earliest in α^O ; this implies that there exists no attribute in our path $D \in P'$ such that $D <_{\mathcal{H},\alpha} C$. Define an operator \oplus'' such that $(C, \oplus'') \in \alpha^O$. Since $\oplus \neq \oplus'$, either $\oplus \neq \oplus''$ or $\oplus' \neq \oplus''$. Without loss of generality, assume that $\oplus \neq \oplus''$.

Consider the subpath of P from A to C . It is shorter than N and connects two attributes with different operators. We apply our inductive hypothesis to get that there exists some $D \in P$ such that either $D <_{\mathcal{H},\alpha} A$ or $D <_{\mathcal{H},\alpha} C$. In the first case, we have found an attribute that satisfies our conditions and we are done. In the second case, we know that $D \notin P'$ by our definition of C . Thus D must be A ; we have that $A <_{\mathcal{H},\alpha} C$.

Consider the subpath of P from C to B ; let X_i denote the i^{th} node in this path for $0 \leq i \leq k$, where $X_0 = C$ and $X_k = B$. We claim that for all $i < k$, $A <_{\mathcal{H},\alpha} X_i$. We argue this inductively; for our base case, we are given that $A <_{\mathcal{H},\alpha} C = X_0$. Now let $i \geq 1$, and assume $A <_{\mathcal{H},\alpha} X_j$ for $j < i$. Call this the *inner inductive hypothesis*.

Note that we have $A <_{\mathcal{H},\alpha} C$ and that C must precede X_i by definition. Thus A precedes X_i in α^O . All that remains is showing that $A <_{\mathcal{H},\alpha} X_i$. Define \oplus^i such that $(X_i, \oplus^i) \in \alpha^O$. Since we assumed earlier that $\oplus \neq \oplus''$, we know that either $\oplus^i \neq \oplus$ or $\oplus^i \neq \oplus''$.

■ $\oplus^i \neq \oplus$

By our (inner) inductive hypothesis, we know that $A <_{\mathcal{H},\alpha} X_{i-1}$. We also know that there must exist some edge $E \in \mathcal{E}$ such that $X_{i-1}, X_i \in E$. Thus by condition i.1, $A \sim_{\mathcal{H},\alpha} X_i$.

■ $\oplus^i \neq \oplus''$

By our (outer) inductive hypothesis, we know that for some $0 \leq j \leq i$, $X_j <_{\mathcal{H},\alpha} C$ or $X_j <_{\mathcal{H},\alpha} X_i$. By our definition of C , the first case is impossible. And by our (inner) inductive hypothesis, we have that $A <_{\mathcal{H},\alpha} X_j$. We thus have that $A <_{\mathcal{H},\alpha} X_j <_{\mathcal{H},\alpha} X_i$, which implies that $A \sim_{\mathcal{H},\alpha} X_i$ by condition i.2.

This gives us that $A <_{\mathcal{H},\alpha} X_{k-1}$. Since there exists an edge $E \in \mathcal{E}$ such that $X_{k-1}, B \in E$, condition i.1 tells us that $A \sim_{\mathcal{H},\alpha} B$. As before, this implies that either $A <_{\mathcal{H},\alpha} B$ or $B <_{\mathcal{H},\alpha} A$. \blacktriangleleft

► **Lemma 65** (Copy of Lemma 24). *Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and an aggregation ordering α , suppose we have two attributes $A, B \in \mathcal{V}(\alpha)$ such that $A <_{\mathcal{H},\alpha} B$. Then there must exist a path P from A to B such that for every $C \in P, C \neq A$ we have $A <_{\mathcal{H},\alpha} C$.*

Proof. Define \oplus and \oplus' such that $(A, \oplus), (B, \oplus') \in \alpha$. In addition define i such that $A <_{\mathcal{H},\alpha}^i B$, which implies that A precedes B in α and that $A \sim_{\mathcal{H},\alpha}^i B$. Our proof is by induction on i . For our basecase, if $A \sim_{\mathcal{H},\alpha}^0 B$, we know that $\exists E \in \mathcal{E} : A, B \in E$. Thus the path $P = AB$ satisfies our conditions.

For $i > 0$, we have the following cases:

- $\oplus \neq \oplus'$ and $\exists E \in \mathcal{E}, (C, \oplus'') \in \alpha^O : B, C \in E, A <_{\mathcal{H}, \alpha}^{i-1} C$
By our inductive hypothesis, there must exist a path P' from A to C such that for all $D \in P', D \neq A$ we have $A <_{\mathcal{H}, \alpha} D$. Then the path $P = P'B$ satisfies our conditions.
- $\oplus \neq \oplus'$ and $\exists E \in \mathcal{E}, (C, \oplus'') \in \alpha^O : A, C \in E, B <_{\mathcal{H}, \alpha}^{i-1} C$
By our inductive hypothesis, there must exist a path P' from B to C such that for all $D \in P', D \neq B$ we have $B <_{\mathcal{H}, \alpha} D$, which also implies that $A <_{\mathcal{H}, \alpha} D$ by i.2. Let $\overline{P'}$ be the reverse of P' . Then the path $P = \overline{A}P'$ satisfies our condition.
- $\exists C \in \mathcal{V}$ and $j, k < i : A <_{\mathcal{H}, \alpha}^j C <_{\mathcal{H}, \alpha}^k B$
By our inductive hypothesis, there must exist two paths P' and P'' . P' is a path from A to C such that for all $D \in P', D \neq A$ we have $A <_{\mathcal{H}, \alpha} D$. Similarly, P'' is a path from C to B such that for all $D \in P'', D \neq C$ we have $C <_{\mathcal{H}, \alpha} D$, which implies $A <_{\mathcal{H}, \alpha} D$. Thus the path $P = P'P''$ satisfies our conditions.

◀

► **Theorem 66** (Copy of Theorem 14). *Suppose we are given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and aggregation orderings α, β . Then $\alpha \equiv_{\mathcal{H}} \beta$ if and only if β is a linear extension of $<_{\mathcal{H}, \alpha}$.*

Proof. Soundness:

We use induction on the number of inversions in β with respect to the ordering α . *Base Case:* 0 inversions. Then β is identical to α and $\alpha \equiv_{\mathcal{H}} \beta$.

Induction: Suppose β has $N > 0$ inversions, and assume the lemma is true for orderings with $< N$ inversions. There must be some β_i and β_{i+1} that are inverted with respect to α . Consider the ordering β' derived by swapping β_i and β_{i+1} . It has $N - 1$ inversions with respect to α and is clearly a linear extension of $<_{\mathcal{H}, \alpha}$. Thus, by the inductive hypothesis, $\alpha \equiv_{\mathcal{H}} \beta'$.

We now show that $\beta \equiv_{\mathcal{H}} \beta'$. Suppose $\beta_i = (A, \oplus)$ and $\beta_{i+1} = (B, \oplus')$. We have two cases to consider.

- $\oplus = \oplus'$
By Theorem 9, we can swap β_i and β_{i+1} without affecting the output. This implies that $\beta \equiv_{\mathcal{H}} \beta'$.
- $\oplus \neq \oplus'$
By Lemma 23 and since we know A and B are incomparable under $<_{\mathcal{H}, \alpha}$, any path between A and B must go through some attribute C such that $C <_{\mathcal{H}, \alpha} A$ or $C <_{\mathcal{H}, \alpha} B$. Since β is a valid linear extension of $<_{\mathcal{H}, \alpha}$, these attributes C appear earlier than index i in β . This implies that A and B are in separate connected components in $\pi_{V(\beta_i, \beta_{i+1}, \dots, \beta_{|\beta|})} \mathcal{H}$, which implies that we can swap β_i and β_{i+1} without affecting the output by Theorem 9. This implies that $\beta \equiv_{\mathcal{H}} \beta'$.

Completeness:

We prove the contrapositive: we assume that we are given aggregation orderings α, β such that β is not a linear extension of $<_{\mathcal{H}, \alpha}$, and we will show that $\alpha \not\equiv_{\mathcal{H}} \beta$. We will do so by constructing an instance \hat{I} such that $Q_{\mathcal{H}, \alpha}(\hat{I}) \neq Q_{\mathcal{H}, \beta}(\hat{I})$.

We assume without loss of generality that $\mathcal{V} = V(\alpha)$, i.e. that there are no output attributes. We will provide an example where β and α must differ in the single annotation that comprises the output. If there are output attributes, we can augment our example by putting 1s in all the output attributes; our output will be composed of a single tuple composed of all 1s with the same annotation as in our example below.

Consider the set of all valid linear extensions of $<_{\mathcal{H}, \alpha}$. Suppose the maximum length prefix identical to the prefix of β is of size k . Among all linear extensions with maximum length identical prefixes, suppose the minimum possible index for β_{k+1} is k' . Consider a linear extension α' such that $\alpha'_i = \beta_i$ for $i \leq k$ and $\alpha'_{k'} = \beta_{k+1}$. By the soundness part of our proof, $\alpha' \equiv_{\mathcal{H}} \alpha$; to show that $\alpha \not\equiv_{\mathcal{H}} \beta$ we can simply show that $\alpha' \not\equiv_{\mathcal{H}} \beta$.

Suppose $\alpha'_{k'} = (A, \oplus) = \beta_{k+1}$ and $\alpha'_{k'-1} = (B, \oplus')$. We know that $B <_{\mathcal{H}, \alpha} A$ since k' is the minimum possible index for β_{k+1} in any linear extension of $<_{\mathcal{H}, \alpha}$. Also, since B and A are adjacent in α' , we know that there cannot exist any C such that $B <_{\mathcal{H}, \alpha} C <_{\mathcal{H}, \alpha} A$. These two facts combine to imply $\oplus \neq \oplus'$. Then, by Lemma 24, there exists a path P from A to B such that every attribute in our path $C \in P$ other than A and B must appear after index k' in α' .

Since $\oplus \neq \oplus'$, there must exist $x, y \in \mathbf{D}$ such that $x \oplus y \neq x \oplus' y$. Define a relation $\widehat{R}_{\mathcal{V}}$ with two tuples. The first tuple will contain a 1 for each attribute and an annotation x . The second tuple will contain a 2 for each attribute in P (including A and B) and a 1 for every other attribute. The second tuple will be annotated with y . Note that among the attributes in P , (A, \oplus) is the outermost aggregation in β and (B, \oplus') is the outermost aggregation in α . This implies that

$$\Sigma_{\alpha'_1} \Sigma_{\alpha'_2} \cdots \Sigma_{\alpha'_{n'}} \widehat{R}_{\mathcal{V}} = x \oplus' y \quad \Sigma_{\beta_1} \Sigma_{\beta_2} \cdots \Sigma_{\beta_{|\beta|}} \widehat{R}_{\mathcal{V}} = x \oplus y$$

Let C be the attribute in P right before B ; by definition there must exist an edge $E \in \mathcal{E}$ such that $B, C \in E$. Consider the following instance over the schema \mathcal{H} :

$$\hat{I} = \{\pi_E \widehat{R}_{\mathcal{V}}\} \cup \{\pi_F^1 \widehat{R}_{\mathcal{V}} | F \in \mathcal{E}, F \neq E\}.$$

By definition, $\bowtie_{R_F \in \hat{I}} R_F = \widehat{R}_{\mathcal{V}}$. Since we know that $x \oplus y \neq x \oplus' y$, we have that

$$\Sigma_{\alpha'_1} \Sigma_{\alpha'_2} \cdots \Sigma_{\alpha'_{n'}} \bowtie_{R_F \in \hat{I}} R_F \neq \Sigma_{\beta_1} \Sigma_{\beta_2} \cdots \Sigma_{\beta_{|\beta|}} \bowtie_{R_F \in \hat{I}} R_F$$

We thus have that $Q_{\mathcal{H}, \alpha'}(\hat{I}) \neq Q_{\mathcal{H}, \beta}(\hat{I})$, which implies that $\alpha' \not\equiv_{\mathcal{H}} \beta$. \blacktriangleleft

D Decomposing Valid GHDs: Proofs

We start by stating and proving a useful lemma about the aggregation orderings seen in the sub-trees of a decomposable GHD.

► **Lemma 67.** *Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and an aggregation ordering α , suppose C is a connected component of $\mathcal{H} \setminus V(-\alpha)$. Define $\mathcal{H}_C = (\bigcup_{E \in \mathcal{E}_C} E, \mathcal{E}_C)$ ⁸. For any $A \in C, B \in \mathcal{V}$, if $A <_{\mathcal{H}, \alpha} B$ then $A <_{\mathcal{H}_C, \alpha_{C \setminus C^O}} B$. Similarly, if $A <_{\mathcal{H}_C, \alpha_C \setminus C^O} B$, either $A \in C^O$ or $A <_{\mathcal{H}, \alpha} B$.*

Proof. First we show that for any $A \in C$, if $A <_{\mathcal{H}, \alpha} B$ then $B \in C \setminus C^O$. If $B \in C^O$, then by definition, $A \not<_{\mathcal{H}, \alpha} B$. If $B \notin C$, then every path between A and C must go through attributes in $V(-\alpha)$. Thus, by the contrapositive of Lemma 24, $A \not<_{\mathcal{H}, \alpha} B$. This implies that $A <_{\mathcal{H}, \alpha} B$ only for $B \in C \setminus C^O$.

Note that for any $A \in C^O$, since A is an output attribute in $\alpha_{C \setminus C^O}$, $A <_{\mathcal{H}_C, \alpha_{C \setminus C^O}} B$ for all $B \in C \setminus C^O$. This proves our lemma for $A \in C^O$.

For $A \in C \setminus C^O$, we prove the lemma by showing that for any $i \geq 0$, $\{B | A <_{\mathcal{H}, \alpha}^i B\} = \{B | A <_{\mathcal{H}_C, \alpha_{C \setminus C^O}}^i B\}$. Note that our earlier result shows that both of these sets are subsets of $C \setminus C^O$, so we know that for any i , any B in either set appears in both aggregation orderings.

Proof by induction on i . We first consider the base case: $i = 0$. We note that since A is not an output attribute condition (0.2) is irrelevant. Since \mathcal{H}_C contains all edges involving attributes in C and $\alpha_{C \setminus C^O}$ preserves the ordering and the operators of elements of α , condition (0.1) applies to the same set of attributes in both AJAR queries $Q_{\mathcal{H}, \alpha}$ and $Q_{\mathcal{H}_C, \alpha_{C \setminus C^O}}$. Thus $\{B | A <_{\mathcal{H}, \alpha}^0 B\} = \{B | A <_{\mathcal{H}_C, \alpha_{C \setminus C^O}}^0 B\}$.

⁸ Recall $\mathcal{E}_C = \{E \in \mathcal{E} | E \cap C \neq \emptyset\}$.

For $i > 0$, the inductive hypothesis supposes $\{B|A <_{\mathcal{H},\alpha}^j B\} = \{B|A <_{\mathcal{H}_C,\alpha_{C \setminus CO}}^j B\}$ for all $j < i$. Again since \mathcal{H}_C contains all edges involving attributes in C and $\alpha_{C \setminus CO}$ preserves the ordering and the operators of elements of α , the inductive hypothesis trivially implies conditions (i.1) and (i.2) apply to the same set of attributes. \blacktriangleleft

► **Theorem 68** (Copy of Theorem 27). *Every decomposable GHD is valid.*

Proof. Suppose the AJAR query is $Q_{\mathcal{H},\alpha}$. We need to show for any A, B such that $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, $A \not<_{\mathcal{H},\alpha} B$. Proof by induction on $|\alpha|$. If $|\alpha| = 0$, all GHDs are valid and decomposable. For $|\alpha| > 0$, we note \mathcal{T}_0 ensure the output attributes are above non-output attributes.

If A and B are non-output attributes and $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, then both are in some \mathcal{T}_C . By the inductive hypothesis, \mathcal{T}_C is valid with respect to $Q_{(\cup_{E \in \mathcal{E}_C} E, \mathcal{E}_C), \alpha_{C \setminus CO}}$. By Lemma 67, this implies $A \not<_{\mathcal{H},\alpha} B$. \blacktriangleleft

► **Theorem 69** (Copy of Theorem 29). *For every valid GHD (\mathcal{T}, χ) , there exists a decomposable GHD (\mathcal{T}', χ') such that for all node-monotone functions γ , the γ -width of (\mathcal{T}', χ') is no larger than the γ -width of (\mathcal{T}, χ) .*

In the proof sketch provided in Section 5, we claim to have width-preserving transformations of a GHD that can enforce two additional properties, which we now present and name:

■ *TOP-unique*: every node $t \in \mathcal{T}$ is $TOP_{\mathcal{T}}(A)$ for exactly one attribute A

■ *subtree-connected*: for any node $t \in \mathcal{T}$ and the subtree \mathcal{T}_t rooted at t , the attributes $\{v \in \mathcal{V} | TOP_{\mathcal{T}}(v) \in \mathcal{T}_t\}$ form a connected subgraph of \mathcal{H}

We first have two lemmas proving the transformations required to enforce these properties are width-preserving.

► **Lemma 70.** *Given a valid GHD (\mathcal{T}, χ) with γ -width w , we can transform it to be TOP-unique while ensuring γ -width $\leq w$.*

Proof. Define a function $TOP_{\mathcal{T}}^{-1} : \mathcal{T} \rightarrow 2^{\mathcal{V}}$ from nodes to sets of attributes such that $TOP_{\mathcal{T}}^{-1}(t) = \{A | TOP_{\mathcal{T}}(A) = t\}$.

First we eliminate nodes $t \in \mathcal{T}$ such that $|TOP_{\mathcal{T}}^{-1}(t)| = 0$. We note, by definition, $\chi(t) \subseteq \chi(\text{parent}(t))$. This implies that we can simply remove t , connecting all of its children to $\text{parent}(t)$ without violating any properties of the valid GHD. And the width is trivially preserved.

Now suppose for some node $t \in \mathcal{T}$, $|TOP_{\mathcal{T}}^{-1}(t)| = k > 1$. Let A_1 be the attribute in $TOP_{\mathcal{T}}^{-1}(t)$ that is earliest in the aggregation ordering. Let $X = \chi(t) \cap \chi(\text{parent}(t))$. Then create a new node t' such that $\chi(t') = \{A_1\} \cup X$ and add it to \mathcal{T} between t and $\text{parent}(t)$. All of properties of the valid GHD must still hold, and since the new node contains a subset of the attributes in t , the width must be preserved. Note that after adding this node, $|TOP_{\mathcal{T}}^{-1}(t)| = k - 1$; we can repeat this process until the set is of size 1. \blacktriangleleft

► **Lemma 71.** *Given a valid GHD (\mathcal{T}, χ) with γ -width w that is TOP-unique, we can transform it to be subtree-connected while preserving TOP-unique and γ -width $\leq w$.*

Proof. For any node $t \in \mathcal{T}$, define $\mathcal{V}_t = \{v \in \mathcal{V} | TOP_{\mathcal{T}}(v) \in \mathcal{T}_t\}$.

We proceed with a proof by (bottom-up) induction on the tree \mathcal{T} . As our base case, we consider the leaves of \mathcal{T} . Since t does not have any children, \mathcal{V}_t must contain exactly one attribute, which is trivially a connected subgraph of \mathcal{H} .

Now we consider the subtree \mathcal{T}_t rooted at some internal node t . Let A be the attribute such that $TOP_{\mathcal{T}}(A) = t$. Let c_1, c_2, \dots, c_k be the children of t . By the inductive hypothesis, the subtrees rooted at these children satisfy all of the desired properties. We note that, by definition,

$\chi(t) \setminus A \subseteq \chi(\text{parent}(t))$. For any child c_i such that \mathcal{V}_{c_i} and A are not connected in \mathcal{H} , we can remove A from $\chi(\mathcal{T}_{c_i})$ and set $\text{parent}(c_i)$ to be $\text{parent}(t)$. By doing so for all such children c_i , we ensure that \mathcal{V}_t is a connected subgraph of \mathcal{H} . Since A is not connected to \mathcal{V}_{c_i} , this transformation does not violate the properties of GHDs. Since we are not creating any new ancestral relationships between nodes, the transformation does not violate the properties of valid GHDs. Finally, the γ -width $\leq w$ and TOP-unique properties are preserved trivially. \blacktriangleleft

We have thus established that we can transform any valid GHD to additionally satisfy TOP-unique and subtree-connected while preserving width. We now show that any valid GHD satisfying the two additional properties is decomposable. Combined with the two lemmas above, this will complete the proof of Theorem 29. Before we dive into the proof, we prove two helpful lemmas.

► **Lemma 72.** *Given an AJAR+ query $Q_{\mathcal{H},\alpha}$, if $A <_{\mathcal{H},\alpha} B$ for A, B with identical operators, there must exist some C with a different operator such that $A <_{\mathcal{H},\alpha} C <_{\mathcal{H},\alpha} B$.*

Proof. $A <_{\mathcal{H},\alpha}^i B$ for some fixed i . If A, B have identical operators, the only way $A <_{\mathcal{H},\alpha}^i B$ is via rule (i.2), which requires some C and $j, k < i$ such that $A <_{\mathcal{H},\alpha}^j C <_{\mathcal{H},\alpha}^k B$. If this C has the same operator as A and B , we can repeatedly apply this rule until we find some attribute between A and B with a different operator (since both of the rules for $i = 0$ only apply to attributes with differing operators). \blacktriangleleft

► **Lemma 73.** *Given an AJAR+ query $Q_{\mathcal{H},\alpha}$ and valid GHD $(\mathcal{T} < \chi)$. Suppose $A <_{\mathcal{H},\alpha} B$, A is not an output attribute, and $\text{TOP}_{\mathcal{T}}(A)$ is a top node only for A . Then, $\text{TOP}_{\mathcal{T}}(A)$ must be an ancestor of $\text{TOP}_{\mathcal{T}}(B)$ in any valid GHD.*

Proof. Lemma 24 implies that there exists a path from A to B such that for every C in the path such that $C \neq A$, $A <_{\mathcal{H},\alpha} C$. Let C_0, C_1, \dots, C_k represent the path, where $C_0 = A$ and $C_k = B$. We claim that $\text{TOP}_{\mathcal{T}}(A)$ is an ancestor of $\text{TOP}_{\mathcal{T}}(C_i)$ for all $1 \leq i \leq k$. Proof by induction on i . Our base case is for $i = 1$. By the definition of a path, A and C_1 must appear together in some hyperedge, implying that they appear together in some bag of \mathcal{T} . Both $\text{TOP}_{\mathcal{T}}(C_1)$ and $\text{TOP}_{\mathcal{T}}(A)$ must either be equal to or an ancestor of this bag. Since $\text{TOP}_{\mathcal{T}}(C_1)$ cannot be equal to or an ancestor of $\text{TOP}_{\mathcal{T}}(A)$, $\text{TOP}_{\mathcal{T}}(A)$ is an ancestor of $\text{TOP}_{\mathcal{T}}(C_1)$.

For $i > 1$, we note since C_i and C_{i-1} appear in an edge together, by the same logic as above, $\text{TOP}_{\mathcal{T}}(C_i)$ and $\text{TOP}_{\mathcal{T}}(C_{i-1})$ must both be equal to or an ancestor of some node $t \in \mathcal{T}$. By the inductive hypothesis, $\text{TOP}_{\mathcal{T}}(A)$ is an ancestor of $\text{TOP}_{\mathcal{T}}(C_{i-1})$, implying that $\text{TOP}_{\mathcal{T}}(A)$ is an ancestor of t . Since $\text{TOP}_{\mathcal{T}}(C_i)$ cannot equal or be an ancestor of $\text{TOP}_{\mathcal{T}}(A)$, $\text{TOP}_{\mathcal{T}}(A)$ must be an ancestor of $\text{TOP}_{\mathcal{T}}(C_i)$. \blacktriangleleft

► **Lemma 74.** *Any valid GHD (\mathcal{T}, χ) that is TOP-unique and subtree-connected must be decomposable.*

Proof. We actually prove a slightly stronger statement. Define the property TOP-semiunique as follows: every non-root node $t \in \mathcal{T}$ is the $\text{TOP}_{\mathcal{T}}$ node for exactly one attribute and the root node is either the $\text{TOP}_{\mathcal{T}}$ for exactly one attribute or more than one output attribute (and zero non-output attributes). Note that the TOP-unique property directly implies the TOP-semiunique property. We will show that if (\mathcal{T}, χ) is a valid, TOP-semiunique, and subtree-connected GHD for the AJAR query $Q_{\mathcal{H},\alpha}$, it must be decomposable.

Proof by induction on $|\alpha|$. If $|\alpha| = 0$, then every GHD is decomposable.

Suppose $|\alpha| > 0$. Consider the set of nodes that are $\text{TOP}_{\mathcal{T}}$ nodes for output attributes, i.e. $\{t \in \mathcal{T} \mid \exists A \in V(-\alpha) : \text{TOP}_{\mathcal{T}}(A) = t\}$. Since no non-output attribute can have a top node above an output attribute's top node, the TOP-semiunique property guarantees that this set of nodes forms a rooted subtree \mathcal{T}_0 of \mathcal{T} such that $\chi(\mathcal{T}_0) = V(-\alpha)$.

Consider the subtrees in $\mathcal{T} \setminus \mathcal{T}_0$. Call them $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_k$. For any \mathcal{T}_i , let \mathcal{V}_i be the attributes that have $TOP_{\mathcal{T}}$ nodes in \mathcal{T}_i , i.e. $\mathcal{V}_i = \{A \in \mathcal{V} \mid TOP_{\mathcal{T}}(A) \in \mathcal{T}_i\}$. None of these \mathcal{V}_i can contain any output attributes, and connected-subtree guarantees that each of the \mathcal{V}_i are connected. Thus, the \mathcal{V}_i must be the connected components of $\mathcal{H} \setminus V(-\alpha)$. So for each connected component C of $\mathcal{H} \setminus V(-\alpha)$, the corresponding subtree \mathcal{T}_C is the subtree \mathcal{T}_i such that $\mathcal{V}_i = C$. Since for any $A \in C$, $TOP_{\mathcal{T}}(A) \in \mathcal{T}_C$, the attributes in C only appear in \mathcal{T}_C . Note that for every edge $E \in \mathcal{E}$, there exists a node $t \in \mathcal{T}$ such that $E \subseteq \chi(t)$. This implies that for every edge $E \in \mathcal{E}_C$, there exists a node $t \in \mathcal{T}_C$ such that $E \subseteq \chi(t)$. As such, we can conclude that each \mathcal{T}_C is a GHD for the hypergraph $(\bigcup_{E \in \mathcal{E}_C} E, \mathcal{E}_C)$.

Define $\mathcal{V}_C = \bigcup_{E \in \mathcal{E}_C} E$. To complete this proof, we now need to show that each \mathcal{T}_C is a decomposable GHD for the AJAR query $Q_{(\mathcal{V}_C, \mathcal{E}_C), \alpha_{C \setminus C^O}}$. By the inductive hypothesis, if \mathcal{T}_C is valid, TOP-semiunique and subtree-connected, it must be decomposable. Note that since \mathcal{T} is TOP-semiunique and subtree-connected, \mathcal{T}_C must also be TOP-semiunique and subtree-connected. We have also established that \mathcal{T}_C is a GHD for $(\mathcal{V}_C, \mathcal{E}_C)$. Thus to finish this proof, we only need to show that for any $A, B \in \mathcal{V}_C$ such that $TOP_{\mathcal{T}_C}(B)$ is an ancestor of $TOP_{\mathcal{T}_C}(A)$, $A \not<_{(\mathcal{V}_C, \mathcal{E}_C), \alpha_{C \setminus C^O}} B$.

For ease of notation, in the rest of this proof we will use $<_C$ to represent $<_{(\mathcal{V}_C, \mathcal{E}_C), \alpha_{C \setminus C^O}}$. We show the contrapositive: if $A <_C B$, $TOP_{\mathcal{T}_C}(B)$ is not an ancestor of $TOP_{\mathcal{T}_C}(A)$. We consider a few cases. If $A \in \mathcal{V}_C \setminus C$, A must be in $V(-\alpha)$, implying $TOP_{\mathcal{T}_C}(A)$ is the root of \mathcal{T}_C . For any $A \in C$, note that $TOP_{\mathcal{T}_C}(A) = TOP_{\mathcal{T}}(A)$. By Lemma 67, for any $A \in C$, if $A <_C B$ then either $A <_{\mathcal{H}, \alpha} B$ or $A \in C^O$. In the former case, the fact that \mathcal{T} is valid ensures $TOP_{\mathcal{T}}(B)$ is not an ancestor of $TOP_{\mathcal{T}}(A)$. For the latter case, assume for contradiction that there exist A, B such that $TOP_{\mathcal{T}}(B)$ is an ancestor of $TOP_{\mathcal{T}}(A)$, $A <_C B$, and $A \in C^O$.

We first claim that, without loss of generality, we can suppose that A and B have different operators. To do so, we show that if A and B have the same operator, there must exist a B' with a different operator such that $TOP_{\mathcal{T}}(B')$ is an ancestor of $TOP_{\mathcal{T}}(A)$ and $A <_C B'$. By the definition of C^O , there must exist some $A' \in C^O$ such that $A' <_{\mathcal{H}, \alpha} B$. Since $A', A \in C$, there must exist a path exclusively in C that connects the two. And since $A', A \in C^O$, no attribute along the path precedes either A or A' in $<_{\mathcal{H}, \alpha}$. The contrapositive of Lemma 23 implies that A' and A must have the same operator, which implies that A' and B have the same operator. Lemmas 72 and 73 imply that there exists some B' with a different operator such that $A' <_{\mathcal{H}, \alpha} B' <_{\mathcal{H}, \alpha} B$ and $TOP_{\mathcal{T}}(B')$ is an ancestor of $TOP_{\mathcal{T}}(B)$. The former result implies $B' \in C \setminus C^O$ and $A <_C B'$. The latter result implies $TOP_{\mathcal{T}}(B')$ is an ancestor of $TOP_{\mathcal{T}}(A)$.

We now suppose A and B have different operators without loss of generality. Since $A \in C^O$, any $O \in \mathcal{V}$ such that $O <_{\mathcal{H}, \alpha} A$ must be an output attribute, thereby implying $O <_{\mathcal{H}, \alpha} B$ as well. This fact, combined with Lemma 23, implies every path between A and B must contain some D such that $D <_{\mathcal{H}, \alpha} B$. Since \mathcal{T} is valid and TOP-semiunique, the $TOP_{\mathcal{T}}(D)$ for each of these D cannot be in the subtree rooted at $TOP_{\mathcal{T}}(B)$. This implies that A and B are disconnected in the subtree rooted at $TOP_{\mathcal{T}}(B)$, contradicting the subtree-connected property. \blacktriangleleft

► **Theorem 75** (Copy of Theorem 31). *For an AJAR query $Q_{\mathcal{H}, \alpha}$, suppose $\mathcal{H}_0, \dots, \mathcal{H}_k$ are the characteristic hypergraphs $H(\mathcal{H}, \alpha)$. Then GHDs G_0, G_1, \dots, G_k of $\mathcal{H}_0, \dots, \mathcal{H}_k$ can be connected to form a decomposable GHD G for $Q_{\mathcal{H}, \alpha}$. Conversely, any decomposable GHD G of $Q_{\mathcal{H}, \alpha}$ can be partitioned into GHDs G_0, G_1, \dots, G_k of the characteristic hypergraphs $\mathcal{H}_0, \dots, \mathcal{H}_k$. Moreover, in both of these cases, $\gamma\text{-width}(G) = \max_i \gamma\text{-width}(G_i)$.*

Proof. Proof by induction on $|\alpha|$. Our base case is $|\alpha| = 0$. In this case, the only characteristic hypergraph is the input hypergraph, that is $H(\mathcal{H}, \alpha) = \mathcal{H}$. The theorem is then trivially true.

Suppose $|\alpha| > 0$. Any decomposable GHD G for $Q_{\mathcal{H}, \alpha}$ must be decomposable into subtrees $\mathcal{T}_0, \dots, \mathcal{T}_l$ such that $\chi(\mathcal{T}_0) = V(-\alpha)$ and \mathcal{T}_i is a decomposable GHD for $Q_{(\bigcup_{E \in \mathcal{E}_{C_i}} E, \mathcal{E}_{C_i}), \alpha_{C_i \setminus C_i^O}}$ where C_i is the i^{th} connected component of $H \setminus V(-\alpha)$. Define \mathcal{V}_{C_i} to be $\bigcup_{E \in \mathcal{E}_{C_i}} E$. To preserve

the running intersection property of a GHD, the root of \mathcal{T}_i and its parent (in \mathcal{T}_0) must contain the attributes $\mathcal{V}_{C_i} \cap V(-\alpha)$. This implies each of the \mathcal{T}_i are decomposable GHDs of $Q_{\mathcal{H}_i^+, \alpha_{C_i \setminus C_i^0}}$, where \mathcal{H}_i^+ is the hypergraph defined in the definition of the characteristic hypergraphs. By the inductive hypothesis, the \mathcal{T}_i (for $i \geq 1$) can be broken down into GHDs G_1, \dots, G_k of the characteristic hypergraphs $\mathcal{H}_1, \dots, \mathcal{H}_k$. In addition, \mathcal{T}_0 must also have nodes that contain the edge $E \in \mathcal{E}$ such that $E \subseteq V(-\alpha)$, implying it is the GHD G_0 of the characteristic hypergraph \mathcal{H}_0 .

In the other direction, by the inductive hypothesis, the GHDs G_1, \dots, G_k can be stitched together to form $\mathcal{T}_1, \dots, \mathcal{T}_l$ such that each \mathcal{T}_i is the decomposable GHD for the AJAR query $Q_{\mathcal{H}_i^+, \alpha_{C_i \setminus C_i^0}}$. Note that, by definition, for each i , \mathcal{T}_i and G_0 must both have a node containing the attributes $\mathcal{V}_{C_i} \cap V(-\alpha)$; let t_i and g_i denote the appropriate node in \mathcal{T}_i and G_0 , respectively. We can re-root \mathcal{T}_i at t_i without violating any conditions since it amounts to re-rooting the top-most GHD of its decomposition; re-rooting \mathcal{T}_i at t_i can only change the ancestor relationship between $TOP_{\mathcal{T}}$ nodes of output attributes. Once we re-root the \mathcal{T}_i appropriately, we can simply set $parent(t_i)$ to be g_i to generate a decomposable GHD for the AJAR query $Q_{\mathcal{H}, \alpha}$. \blacktriangleleft

E Product Aggregations (Detailed version)

The primary application of queries with multiple aggregations is to establish bounds for the Quantified Conjunctive Query (QCQ) problem [15]. A QCQ query consists of an arbitrary conjunctive query preceded by a series of (existential and/or universal) quantifiers, and a solution must report the satisfying assignments to the non-quantified variables. A $\#QCQ$ query is similar to a QCQ query, but instead of reporting satisfying assignments, we report the number of satisfying assignments.

We now introduce a new type of aggregation, called product aggregation, that lets us efficiently handle QCQ queries. We define the AJAR problem for product aggregations, and then extend our algorithm from Section 3.3 to handle this new type of AJAR query.

E.1 Ajar queries with product aggregates

In order to recover QCQ as an AJAR query, we need *product aggregations* i.e. aggregations that use the \otimes operator. Throughout the paper, we have assumed that an absent tuple effectively has an annotation of 0. To maintain this for product aggregations, we need to define product aggregation so that it returns 0 if any tuple is absent. In particular, we redefine $\sum_{(A, \otimes)} R_F$ to include a projected tuple $t_{F \setminus A}$ in the output only if $(t_{F \setminus A} \circ t_A)$ exists in R_F for *every* possible value $t_A \in \mathcal{D}^A$. More formally, let $B = F \setminus A$:

$$\blacktriangleright \text{Definition 76. } \sum_{(A, \otimes)} R_{AB} = \{(t_B, \lambda) : \forall t_A \in \mathcal{D}^A, t_B \circ t_A \in R_{AB} \text{ and } \lambda = \prod_{(t, \lambda_t) \in R_{AB} : \pi_B t = t_B} \lambda_t\}$$

Note that this adjusted definition implies an annotation of 0 is once again fully equivalent to absence. We can adjust the definition of aggregation orderings (and AJAR queries) to possibly include this new type of aggregation. We can construct valid GHDs for such aggregations as before, and run AggroGHDJoin to solve them.

Example 77. Consider the semiring $(\{0, 1\}, \max, \cdot)$. Note that in this domain \max is equivalent to a disjunction (and the logical existential quantifier) and \prod is equivalent to a conjunction (and the logical universal quantifier). Thus the space of AJAR queries that use these two aggregators recover all QCQ queries.

An aggregation using \otimes is called a *product aggregation*, and an attribute that is aggregated using a product aggregation is called a *product attribute*. Aggregations that are not product

aggregations are called *semiring aggregations*, while attributes that are neither output attributes nor product attributes are called *semiring attributes*.

Idempotence Assumption: Using the product aggregation as defined raises one issue. Our semiring aggregates satisfy the distributive property, which is integral in our ability to push-down aggregations and for our results about commuting aggregations (Theorem 9). In general, product aggregations do not distribute: $(a \otimes b) \otimes (a \otimes c) = (a \otimes a) \otimes (b \otimes c) \neq a \otimes (b \otimes c)$. However if we require our product aggregations to be idempotent, that is that $a \otimes a = a$ for any element a , our product aggregations will distribute. And for QCQ , the domain is restricted to $\{0, 1\}$, in which product aggregations are idempotent. So in this section, we will study idempotent product aggregations; we will generalize to non-idempotent aggregations in Appendix E.4.

E.2 Solving Ajar queries with product aggregates

For aggregation orderings that have product aggregations, the rules for determining when two orderings are equivalent are somewhat different. We now discuss how we can optimize this new type of aggregation further; product aggregations are fundamentally different from ordinary aggregation because we can do the aggregation *before* the join, as seen in the following example:

► **Example 78.** In the semiring $(\{0, 1\}, \max, \cdot)$, suppose we have two relations $R(A, B) = \{((0, 0), x), ((0, 1), y)\}$ and $S(B, C) = \{((0, 1), p), ((1, 1), q)\}$. Consider the AJAR query $\sum_{(B, \cdot)} R(A, B) \bowtie S(B, C)$. If we compute the join, we will get two tuples with the annotations $x \cdot p$ and $y \cdot q$, and then aggregating over B will produce a relation with the element $((0, 1), x \cdot p \cdot y \cdot q)$. However, note that $x \cdot p \cdot y \cdot q = (x \cdot y) \cdot (p \cdot q)$, implying that $\sum_{(B, \cdot)} R(A, B) \bowtie S(B, C) = (\sum_{(B, \cdot)} R(A, B)) \bowtie (\sum_{(B, \cdot)} S(B, C))$.

Now we describe our algorithm for solving AJAR queries when product aggregations are present. Our algorithm follows the same lines as the algorithm from Section 3.3. Recall that the algorithm consisted of searching for *equivalent orderings*, then searching for GHD *compatible* with an equivalent ordering, and running AggroGHDJoin on the GHD with the smallest fhw. For product aggregations, we need to modify our algorithm for testing equivalent orderings, and our definition of compatibility; we do these in turn.

Testing orderings for equivalence

Algorithm 7 gives the pseudo-code for our equivalence test for orderings containing product aggregates.

We have the lemma analogous to Lemma 10.

► **Lemma 79** (Copy of Lemma 44). *Algorithm 7 returns True if and only if $\alpha \equiv_{\mathcal{H}} \beta$.*

Proof. Soundness: Suppose Algorithm 7 returns true; we will show $\alpha \equiv_{\mathcal{H}} \beta$. We induct on the length $|\alpha|$. For our base case, when $|\alpha| = 0$, we return true when $|\beta| = 0$. In this case, the two (empty) orderings are trivially equivalent.

Suppose $|\alpha| > 0$. We have two cases: when $\mathcal{H} \setminus (V(-\alpha) \cup \text{PA}(\alpha))$ has one component and when it has multiple components. We first consider the multiple components case. Let the components be C_1, \dots, C_m . Then we define C'_1, \dots, C'_m as in the algorithm i.e. For $1 < i < m$, let \mathcal{E}_i be $\{E \in \mathcal{E} | E \cap C_i \neq \emptyset\}$, the elements of \mathcal{E} that intersect with C_i . Then $C'_i = C_i \cup \bigcup_{E \in \mathcal{E}_i} E \cap \text{PA}(\alpha)$. We define \mathcal{E}_0 to be $\mathcal{E} \setminus (\bigcup_{1 \leq i < m} \mathcal{E}_i)$ (these are relations with only output attributes or product aggregations). Accordingly let C'_0 be the product aggregations that appear in \mathcal{E}_0 . We can then express the following identities:

$$\begin{aligned} \bowtie_{F \in \mathcal{E}} R_F &= \bowtie_{0 \leq i \leq m} \bowtie_{F \in \mathcal{E}_i} R_F \\ \sum_{\alpha} \bowtie_{F \in \mathcal{E}} R_F &= \bowtie_{0 \leq i \leq m} \sum_{\alpha_{C'_i}} \bowtie_{F \in \mathcal{E}_i} R_F \end{aligned}$$

Algorithm 7 TestEquivalence($\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, α, β)**Input:** Query hypergraph \mathcal{H} , orderings α, β .**Output:** True if $\alpha \equiv_{\mathcal{H}} \beta$, False otherwise.

```

if  $|\alpha| = |\beta| = 0$  then
    return True
end if
Remove  $V(-\alpha)$  from  $\mathcal{H}$ , then divide  $\mathcal{H}$  into connected components  $C_1, \dots, C_m$ .
if  $m > 1$  then
    return  $\wedge_i \text{TestEquivalence}(\mathcal{H}, \alpha_{C_i}, \beta_{C_i})$ 
end if
Choose  $j$  such that  $\beta_j = \alpha_1$ . Let  $\beta_j = (b_j, \oplus'_j)$ .
if  $\exists i < j : \beta_i = (b_i, \oplus'_i)$ ,  $\oplus'_i \neq \oplus'_j$  and there is a path from  $b_i$  to  $b_j$  in  $\{b_i, b_{i+1}, \dots, b_{|\alpha|}\}$ 
then
    return False
end if
Let  $\beta'$  be  $\beta$  with  $\beta_j$  removed.
Let  $\alpha'$  be  $\alpha$  with  $\alpha_1$  removed.
return TestEquivalence( $\mathcal{H}, \alpha', \beta'$ )

```

The RHS may have a product aggregation (a, \otimes) happening in multiple components, but it happens exactly once per relation containing a . We note this identity holds for β as well. This identity implies that $\alpha \equiv_{\mathcal{H}} \beta$ if $\alpha_{C'_i} \equiv_{\mathcal{H}} \beta_{C'_i}$ for all i . We note that for $i = 0$, all of the aggregations contain the same operator, so any ordering is equivalent. For $i > 0$, we note that we return true only if all of the recursive calls return true, implying $\alpha_{C'_i} \equiv_{\mathcal{H}} \beta_{C'_i}$ by the inductive hypothesis.

When $\mathcal{H} \setminus (V(-\alpha) \cup \text{PA}(\alpha))$ has one component, we choose j such that $\beta_j = \alpha_1$ and define β' to be β with β_j removed. Note α' is defined to be α with $\alpha_1 = \beta_j$ removed. To show $\beta \equiv_{\mathcal{H}} \alpha$, we need to show $\alpha' \equiv_{\mathcal{H}} \beta'$ and $\beta \equiv_{\mathcal{H}} \beta_j \beta'$. Since we return true only when our recursive call on α' and β' returns true, the former equivalence holds by the inductive hypothesis.

To show $\beta \equiv_{\mathcal{H}} \beta_j \beta'$, we ensure β_j and β_i can commute for all $i < j$. More specifically, we ensure that if β_j can be moved to index $i + 1$, it can be moved to index i . For any β_i with the same operator, β_i and β_j trivially commute. If β_i has a different operator, we know there is no path between their attributes b_i and b_j among the nodes

$$(\{b_i, b_{i+1}, \dots, b_{|\alpha|}\} \setminus \text{PA}(\alpha)) \cup \{b_i, b_j\}.$$

Let V be this set of attributes. Define $V_1 \subset V$ to be the set of nodes connected to b_i in the hypergraph restricted to V (we know $b_j \notin V_1$). Let \mathcal{E}_1 be the set of edges that contain some attribute in V_1 , i.e. $\{E \in \mathcal{E} | E \cap V_1 \neq \emptyset\}$. We note that the attributes of $V \setminus V_1$ do not appear in the edges of \mathcal{E}_1 . Let $\mathcal{E}_2 = \mathcal{E} \setminus \mathcal{E}_1$; the attributes of $V \setminus V_1$ all appear in \mathcal{E}_2 . We can then express the following identities:

$$\begin{aligned}
\bowtie_{F \in \mathcal{E}} R_F &= (\bowtie_{F \in \mathcal{E}_1} R_F) \bowtie (\bowtie_{F \in \mathcal{E}_2} R_F) \\
\sum_{\beta_{V \cup \text{PA}(\alpha)}} \bowtie_{F \in \mathcal{E}} R_F &= \left(\sum_{\beta_{V_1 \cup \text{PA}(\alpha)}} \bowtie_{F \in \mathcal{E}_1} R_F \right) \bowtie \\
&\quad \left(\sum_{\beta_{V_2 \cup \text{PA}(\alpha)}} \bowtie_{F \in \mathcal{E}_2} R_F \right)
\end{aligned}$$

We note, by definition, that β_i and β_j must be pushed down into different aggregations in the

previous expression. This implies that we can commute β_i and β_j when they are adjacent, completing the soundness proof.

Completeness: We prove that if Algorithm 7 returns false, then there must exist a database instance I such that $Q_{\mathcal{H},\alpha}(I) \neq Q_{\mathcal{H},\beta}(I)$.

If Algorithm 7 returns false, there must be a component C' , $\alpha' = \alpha_{C'}$, $\beta' = \beta_{C'}$, such that $\beta_j = \alpha_1$, and there exists a $i < j$ such that $\beta_i = (b_i, \odot'_i)$, $\beta_j = (b_j, \odot'_j)$, $\odot'_i \neq \odot'_j$ and there is a path from b_i to b_j that consists of only b_i, b_j , and semiring attributes in $\{b_i, b_{i+1}, \dots, b_{|\alpha'|}\}$. We now define our instance I that gives different outputs on these orderings.

If neither \odot'_i nor \odot'_j are product operators, then choose x, y such that $x \odot'_i y \neq x \odot'_j y$. If one of them is a product operator while the other is not, choose $x = y = 1$. Now we define the attribute domains. Let B be the set of attributes in the path from b_i to b_j consisting of b_i, b_j and semiring attributes in $\{b_i, b_{i+1}, \dots, b_{|\alpha'|}\}$. For every $b \in B$, we set $\mathcal{D}^b = \{0, 1\}$. For every $b' \notin B$, we set its $\mathcal{D}^{b'}$ to $\{0\}$. In every relation that has at least one attribute from B , it has two tuples. One tuple has value 0 for all attributes in B , the other has value 1 for all attributes in B . The values of the other attributes are of course always 0. One of the relations containing a attribute from B has annotation x for the tuple with 0s and annotation y for the tuple with 1s. All other annotations are 1.

Clearly, each aggregation for an attribute $b' \notin B$ is a no-op, since the domain size $|\mathcal{D}^{b'}| = 1$. Moreover, all aggregations other than β_i, β_j in β and α are also no-ops, because they are non-product aggregations (from the way we chose B) and there is a unique value of the attribute for each tuple it maps to after aggregation.

Thus if both β_i and β_j are non-product aggregations themselves, then we have $Q_{\mathcal{H},\alpha}(I) = x \odot'_j y, Q_{\mathcal{H},\beta}(I) = x \odot'_i y$ which are unequal due to how we chose x and y . If one of them, say β_j is a product aggregation, then $Q_{\mathcal{H},\alpha}(I) = 1$ while $Q_{\mathcal{H},\beta}(I) = 0$ (and vice versa if β_i is a product aggregation). This is because in β , when we do the product aggregation β_j , there is only one value of b_j per corresponding output value, so the product annotation is 0 (and finally the β_i aggregation adds two 0's to get 0). On the other hand, for α , $\beta_j = \alpha_1$ happens when b_j has two values 0, 1 corresponding to a single output tuple, so their annotations are multiplied to get $x \otimes y = 1$. This shows that Algorithm 7 is complete. \blacktriangleleft

Compatible GHDs

Product aggregations not only change the set of equivalent orderings, but also the set of GHDs compatible with a given ordering. In fact, product aggregations allow us to break the rules of GHDs without causing incorrect behavior. In particular, we can have a product attribute P appear in completely disparate parts of the GHD. Thus before defining compatibility for GHDs, we define the notion of *product partitions*.

► **Definition 80.** Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and aggregation ordering α , let $S = \{a \in \mathcal{V} | (a, \otimes) \in \alpha\}$ be the set of attributes with product aggregations. A *product partition* is a set $\{P_a | a \in S\}$ where P_a is a partition of $\{F \in \mathcal{E} | a \in F\}$ (the relations that contain a).

We will duplicate each attribute a for each partition of P_a and have the partition specify which edges contain each instance of a .

► **Definition 81.** Suppose we are given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, aggregation ordering α , and product partition P . The *product partition hypergraph* \mathcal{H}_P is the pair $(\mathcal{V}_P, \mathcal{E}_P)$ such that

- $S = \{a \in \mathcal{V} | P_a \in P\}$
- $\mathcal{V}_P = (\bigcup_{a \in S} \{a_1, a_2, \dots, a_{|P_a|\}}) \cup \mathcal{V} \setminus S$
- $p : \mathcal{V} \times \mathcal{E} \rightarrow \mathcal{V}_P$ where $p(a, F) = a$ if $a \notin S$ otherwise
 a_i where F is in i^{th} partition of P_a
- $\mathcal{E}_P = \bigcup_{F \in \mathcal{E}} \{p(a, F) | a \in F\}$

► **Definition 82.** Given a hypergraph \mathcal{H} and aggregation ordering α , an *aggregating generalized hypertree decomposition* (AGHD) is a triple (\mathcal{T}, χ, P) such that (\mathcal{T}, χ) is a GHD of the product partition hypergraph \mathcal{H}_P .

For any attribute a in the AJAR query, $TOP_{\mathcal{T}}(a)$ for an AGHD (\mathcal{T}, χ, P) can be defined as the set $\{TOP_{\mathcal{T}}(a_1), TOP_{\mathcal{T}}(a_2), \dots, TOP_{\mathcal{T}}(a_{|P_a|})\}$. Now we can define the notion of compatibility of an AGHD, with an ordering.

► **Definition 83.** A AGHD (\mathcal{T}, χ, P) for an AJAR query $Q_{\mathcal{H}, \alpha}$ is compatible with an ordering $\beta \equiv_{\mathcal{H}} \alpha$ if for each attribute pair a, b for which there exists $v_1 \in TOP_{\mathcal{T}}(a)$, $v_2 \in TOP_{\mathcal{T}}(b)$ such that v_1 is an ancestor of v_2 , a must occur before b in the ordering β .

Solving Ajar queries with product aggregates

In our proofs and discussions for the remainder of this section, we will treat the set of $TOP_{\mathcal{T}}$ as a single element for convenience, implicitly placing an existential quantifier before the statement. For example, when we say $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, we mean $\exists t_A \in TOP_{\mathcal{T}}(A), t_B \in TOP_{\mathcal{T}}(B)$ such that t_A is an ancestor of t_B . We also often omit the partition P when referring to an AGHD $G = (\mathcal{T}, \chi, P)$; the partition P can be uniquely defined by (\mathcal{T}, χ) , so we will always assume it is defined appropriately.

We can now modify our algorithm from Section 3.3 to detect equivalent orderings using Algorithm 7, then search for compatible AGHDs, and run AggroGHDJoin over the compatible AGHD with the smallest fhw. Our runtime is given by the next theorem. Note that any AGHD of the original hypergraph is also a GHD of some product partition hypergraph.

► **Theorem 84** (Copy of Theorem 45). *Given a AJAR query $Q_{\mathcal{H}, \alpha}$ possibly involving idempotent product aggregates, let w^* be the smallest fhw for an AGHD compatible with an ordering equivalent to α . Then the runtime for our algorithm is $\tilde{O}(\text{IN}^{w^*} + \text{OUT})$.*

The theorem is proved in Appendix A.

Decomposing AGHDs

We can apply the ideas from Section 5 to AJAR queries with product aggregates as well. In this section we will assume without loss of generality that for any relation R_F , the last aggregation in α_F is not a product aggregation. Suppose this assumption is violated, i.e. there exists some relation R_F such that the last aggregation in α_F is the product aggregation (A_P, \otimes) . We can then immediately perform this aggregation, transforming the relation to $R_{F \setminus \{A_P\}}$ and removing the product aggregation. This assumption ensures that every relation appears in one of the subtrees in the decomposition defined below. We now define some terms.

Given an AJAR query $Q_{\mathcal{H}, \alpha}$, suppose we have a subset of the nodes $V \in \mathcal{V}$. Define \mathcal{E}_V to be $\{E \in \mathcal{E} | E \cap V \neq \emptyset\}$, i.e. the set of edges that intersect with V . Additionally, define $\alpha_{-[i]}$ to be α with the first i elements removed. We will be looking at the connected components of $\mathcal{H} \setminus (V_{-\alpha} \cup \text{PA}(\alpha))$. For any connected component C , let $C^+ = C \cup \{v \in \text{PA}(\alpha) | \exists E \in \mathcal{E}_C : v \in E\}$. Additionally, given an ordering α , we define α^O based on a conditional: if α_1 is a product aggregation, let α^O be just α_1 ; if α_1 is not a product aggregation, let α^O be the set of attributes that can be commuted to the beginning of the ordering. To be more precise for this second case, given an attribute A that appears in α_j with operator \odot , $A \in \alpha^O$ if for all $\alpha_i = (B, \odot')$ such that $i < j$ either $\odot' = \odot$ or A and B are not connected among the nodes $(\alpha_{-[i-1]} \setminus \text{PA}(\alpha_{-[i-1]})) \cup \{A, B\}$.

► **Definition 85.** Given an AJAR query $Q_{\mathcal{H}, \alpha}$, we say an AGHD (\mathcal{T}, χ, P) is *decomposable* if:

- There exists a rooted subtree \mathcal{T}_0 of \mathcal{T} such that $\chi(\mathcal{T}_0) = \mathcal{V}(-\alpha)$ (i.e. output attributes).

- For each connected component C of $\mathcal{H} \setminus (V_{-\alpha} \cup \text{PA}(\alpha))$, there is exactly one subtree $\mathcal{T}_C \in \mathcal{T} \setminus \mathcal{T}_0$ such that \mathcal{T}_C is a decomposable AGHD of $Q_{(\cup_{E \in \mathcal{E}_C} E, \mathcal{E}_C), \alpha_{C+} \setminus \alpha_{C+}^O}$.

Then we have theorems analogous to theorems 27, 29, and 31.

► **Theorem 86** (Copy of Theorem 46). *All decomposable AGHDs are compatible with an ordering β such that $\beta \equiv_{\mathcal{H}} \alpha$.*

Proof. Suppose we are given an AJAR query $Q_{\mathcal{H}, \alpha}$ and a decomposable AGHD G for this query. We show a stronger statement: all decomposable AGHDs are compatible with an ordering β such that $\beta \equiv_{\mathcal{H}} \alpha$ and $\text{PA}(\beta) = \text{PA}(\alpha)$ (i.e. the order of the product attributes does not change). Proof by induction on $|\alpha|$. When $|\alpha| = 0$, all GHDs are decomposable and all GHDs are compatible with α .

Suppose $|\alpha| > 0$. By definition, there is a subtree \mathcal{T}_0 of G such that $\chi(\mathcal{T}_0) = V(-\alpha)$. And for each connected component C of $\mathcal{H} \setminus (V(-\alpha) \cup \text{PA}(\alpha))$, we have a subtree \mathcal{T}_C that is a decomposable GHD for the query $Q_{(\cup_{E \in \mathcal{E}_C} E, \mathcal{E}_C), \alpha_{C+} \setminus \alpha_{C+}^O}$. We will use \mathcal{V}_C to denote $\cup_{E \in \mathcal{E}_C} E$ and \mathcal{H}_C to denote $(\mathcal{V}_C, \mathcal{E}_C)$. Similarly, we will use α^C to represent $\alpha_{C+} \setminus \alpha_{C+}^O$. By the inductive hypothesis, each of these subtrees \mathcal{T}_C is compatible with some ordering β^C such that $\beta^C \equiv_{\mathcal{H}_C} \alpha^C$ and $\text{PA}(\beta^C) = \text{PA}(\alpha^C)$. Note that $\beta^C \equiv_{\mathcal{H}_C} \alpha^C$ trivially implies $\beta^C \equiv_{\mathcal{H}} \alpha^C$.

For each C we will construct a β^{C+} such \mathcal{T}_C is compatible with β^{C+} , $\beta^{C+} \equiv_{\mathcal{H}} \alpha_{C+}$, and $\text{PA}(\beta^{C+}) = \text{PA}(\alpha_{C+})$. Since $\alpha^C = \alpha_{C+} \setminus \alpha_{C+}^O$, this requires adding the elements of α_{C+}^O to β^C . Define β^O to be some ordering of the elements compatible with G (i.e. for any $A, B \in V(\alpha_{C+}^O)$ if $\text{TOP}_{\mathcal{T}_C}(A)$ is an ancestor of $\text{TOP}_{\mathcal{T}_C}(B)$, A precedes B in β^O). We claim the ordering $\beta^{C+} = \beta^O \circ \beta^C$ satisfies our three conditions.

The first condition is that \mathcal{T}_C is compatible with this β^{C+} . This is trivially true because we constructed the ordering by adding output attributes to the start of β^C , with which \mathcal{T}_C is already compatible, in an order that is guaranteed to be compatible.

The second condition is that $\beta^{C+} \equiv_{\mathcal{H}} \alpha_{C+}$. By the definition of α_{C+}^O , $\alpha_{C+} \equiv_{\mathcal{H}} \alpha_{C+}^O \circ \alpha^C$. We know $\beta^C \equiv_{\mathcal{H}} \alpha^C$ by the inductive hypothesis. And we claim $\beta^O \equiv_{\mathcal{H}} \alpha_{C+}^O$, which implies $\beta^{C+} \equiv_{\mathcal{H}} \alpha_{C+}$ by definition. We show this claim by showing that the operators of α_{C+}^O are uniform, implying that its elements can be reordered freely. In particular, consider the first element (A_1, \odot_1) of α_{C+} . Since C^+ is a connected component, there must exist a path between A_1 and every other node among the nodes C^+ . Thus, for any $(B, \odot') \in \alpha_{C+}$ such that $\odot' \neq \odot_1$, A_1 will violate the path condition for commuting and ensure $B \notin V(\alpha_{C+}^O)$.

The third condition is that $\text{PA}(\beta^{C+}) = \text{PA}(\alpha_{C+})$. By the inductive hypothesis, $\text{PA}(\beta^C) = \text{PA}(\alpha^C)$. We simply need to show $\text{PA}(\beta^{C+}) = \text{PA}(\alpha_{C+}^O)$. There are two cases to consider, from the definition of α_{C+}^O . In the first case, both β^{C+} and α_{C+}^O contains only one (product) aggregation. In the second case, the two orderings have no product aggregations. In either case, $\text{PA}(\beta^{C+}) = \text{PA}(\alpha_{C+}^O)$ trivially.

We now need to combine the β^{C+} for each C to construct the desired ordering β as desired. We construct β by repeating the two following steps algorithm until every β^{C+} is empty: (1) remove the non-product output prefixes of β^{C+} and append them to β (interleaved arbitrarily) and (2) remove the earliest remaining product aggregation of $\text{PA}(\alpha)$ from the start of the appropriate β^{C+} and append it to β . Note that this procedure ensures $\beta_{C+} = \beta^{C+}$ for each C , which implies $\beta_{C+} \equiv_{\mathcal{H}} \alpha_{C+}$ and (by the soundness of Algorithm 7) $\beta \equiv_{\mathcal{H}} \alpha$. Also note that the procedure preserves the ordering of the product aggregates, so $\text{PA}(\beta) = \text{PA}(\alpha)$. Finally, the given AGHD G must be compatible with β . The construction of G ensure the top nodes of output attributes are all above the top nodes of non-output attributes, and the top nodes of non-output attributes are in the subtrees \mathcal{T}_C , which means the fact that $\beta_{C+} = \beta^{C+}$ ensures these top nodes are ordered in a compatible manner. ◀

► **Theorem 87** (Copy of Theorem 47). *For every valid AGHD (\mathcal{T}, χ) , there exists a decomposable (\mathcal{T}', χ') such that for all node-monotone functions γ , the γ -width of (\mathcal{T}', χ') is no larger than the γ -width of (\mathcal{T}, χ) .*

Proof. We first modify the definition of subtree-connected from Appendix D:

- *subtree-connected*: for any node $t \in \mathcal{T}$ and the subtree \mathcal{T}_t rooted at t , consider the set the attributes $V_t = \{v \in \mathcal{V} | \text{TOP}_{\mathcal{T}}(v) \in \mathcal{T}_t\}$; we require for any two attributes $A, B \in V_t$, there exists a path from A to B in the set $(V_t \setminus \text{PA}(\alpha)) \cup \{A, B\}$.

This same transformation described Lemma 71 can be used for this adjusted definition. Note that this transformation ensures that any node that is $\text{TOP}_{\mathcal{T}}$ for a product aggregation has only one child. Also note that the described transformation might change the partition function P of the AGHD, but it does not change the compatible order.

Suppose the given AJAR problem is $Q_{\mathcal{H}, \alpha}$. Since (\mathcal{T}, χ) is valid, there must exist an ordering β such that (\mathcal{T}, χ) is compatible with β and $\text{alpha} \equiv_{\mathcal{H}} \beta$. The width-preserving transformations of Appendix D preserve the compatibility with an ordering. So we can apply them to get a TOP-unique and subtree-connected AGHD (\mathcal{T}', χ') that is compatible with β and has γ -width no larger than that of (\mathcal{T}, χ) . We claim that this AGHD is decomposable.

As in Appendix D, we prove that any valid, TOP-semiunique, and subtree-connected GHD for an is decomposable. Proof by induction on $|\alpha|$. If $|\alpha| = 0$, then every GHD is decomposable.

Suppose $|\alpha| > 0$. Consider the set of nodes that are $\text{TOP}_{\mathcal{T}}$ nodes for output attributes, i.e. $\{t \in \mathcal{T} | \exists A \in V(-\alpha) : \text{TOP}_{\mathcal{T}}(A) = t\}$. Since (\mathcal{T}', χ') is compatible with β , no non-output attributes can have a top node above an output attributes top node. Thus, the TOP-semiunique property guarantees that this set of nodes forms a rooted subtree \mathcal{T}_0 of \mathcal{T} such that $\chi(\mathcal{T}_0) = V(-\alpha)$.

Consider the subtrees in $\mathcal{T} \setminus \mathcal{T}_0$. Call them $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_k$. For any \mathcal{T}_i , let \mathcal{V}_i be the attributes that have $\text{TOP}_{\mathcal{T}}$ nodes in \mathcal{T}_i , i.e. $\mathcal{V}_i = \{A \in \mathcal{V} | \text{TOP}_{\mathcal{T}}(A) \in \mathcal{T}_i\}$. None of these \mathcal{V}_i can contain any output attributes, and connected-subtree guarantees that each of the \mathcal{V}_i are connected. Thus, the \mathcal{V}_i must be the C_i^+ as defined earlier. So for each connected component C of $\mathcal{H} \setminus (V(-\alpha) \cup \text{PA}(\alpha))$, the corresponding subtree \mathcal{T}_C is the subtree \mathcal{T}_i such that $\mathcal{V}_i = C^+$. Since for any $A \in C$, $\text{TOP}_{\mathcal{T}}(A) \in \mathcal{T}_C$, the attributes in C only appear in \mathcal{T}_C . Note that for every edge $E \in \mathcal{E}$, there exists a node $t \in \mathcal{T}$ such that $E \subseteq \chi(t)$. This implies that for every edge $E \in \mathcal{E}_C$, there exists a node $t \in \mathcal{T}_C$ such that $E \subseteq \chi(t)$. As such, we can conclude that each \mathcal{T}_C is a GHD for the hypergraph $(\bigcup_{E \in \mathcal{E}_C} E, \mathcal{E}_C)$.

Define $\mathcal{V}_C = \bigcup_{E \in \mathcal{E}_C} E$. To complete this proof, we now need to show that each \mathcal{T}_C is a decomposable GHD for the AJAR query $Q_{(\mathcal{V}_C, \mathcal{E}_C), \alpha_{C^+} \setminus \alpha_{C^+}^O}$. By the inductive hypothesis, if \mathcal{T}_C is valid, TOP-semiunique and subtree-connected, it must be decomposable. Note that since \mathcal{T} is TOP-semiunique and subtree-connected, \mathcal{T}_C must also be TOP-semiunique and subtree-connected. We have also established that \mathcal{T}_C is a GHD for $(\mathcal{V}_C, \mathcal{E}_C)$. Thus to finish this proof, we only need to show that there exists an ordering β' such that $\beta' \equiv_{(\mathcal{V}_C, \mathcal{E}_C)} \alpha_{C^+} \setminus \alpha_{C^+}^O$ and \mathcal{T}_C is compatible with β' .

We know \mathcal{T} is compatible with β and $\beta \equiv \alpha$. We set $\beta' = \beta_{C^+ \setminus \beta_{C^+}^O}$; this implies that $\beta' \equiv_{\mathcal{H}} \alpha_{C^+ \setminus \alpha_{C^+}^O}$ since left hand and right hand sides are simply sub-orderings of *beta* and α , respectively. Furthermore, this implies $\beta' \equiv_{(\mathcal{V}_C, \mathcal{E}_C)} \alpha_{C^+ \setminus \alpha_{C^+}^O}$, as $(\mathcal{V}_C, \mathcal{E}_C)$ is simply \mathcal{H} with some output attributes (of β') removed.

We now need to show that \mathcal{T}_C is compatible with β' . In other words, we need to show for any two attributes $A, B \in \mathcal{V}_C$, if $\text{TOP}_{\mathcal{T}_C}(A)$ is an ancestor of $\text{TOP}_{\mathcal{T}_C}(B)$, either A is an output attribute or A precedes B in β' . We show the contrapositive: if A is not an output attribute and A does not precede B in β' , then $\text{TOP}_{\mathcal{T}_C}(A)$ is not an ancestor of $\text{TOP}_{\mathcal{T}_C}(B)$. There are a couple of cases to consider. If $B \in \mathcal{V}_C \setminus C^+$, B must be in $V(-\alpha)$, implying $\text{TOP}_{\mathcal{T}_C}(B)$ is the

root of \mathcal{T}_C . We note that for attributes in C^+ , $TOP_{\mathcal{T}_C}$ and $TOP_{\mathcal{T}}$ are equivalent, so we use them interchangeably. If $B \in C^+ \setminus \beta_{C+}^O$, then we know B must precede A in β' , which implies B precedes A in β . The fact that \mathcal{T} is compatible with B implies $TOP_{\mathcal{T}}(A)$ is not an ancestor of $TOP_{\mathcal{T}}(B)$. The final case to consider is $B \in \beta_{C+}^O$.

Even in this case, we have two cases to consider, based on the two definitions of β_{C+}^O . If B has a product aggregation, then B must be the first element of β_{C+} . This implies B precedes A in β , guaranteeing that $TOP_{\mathcal{T}}(A)$ is not an ancestor of $TOP_{\mathcal{T}}(B)$. The other case is a bit more involved.

Assume for contradiction that there exist A, B such that $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, $B \in \beta_{C+}^O$, and $A \in \beta'$. We first claim that, without loss of generality, we can suppose that A and B have different operators. To do so, we show that if A and B have the same operator, there must exist a $A' \in \beta'$ with a different operator such that $TOP_{\mathcal{T}}(A')$ is an ancestor $TOP_{\mathcal{T}}(B)$. The fact that $A \notin \beta_{C+}^O$ implies there is an attribute A' with a different operator such that there exists a path between A' and A composed of attributes that appear after A' in β_{C+} . We claim $TOP_{\mathcal{T}}(A')$ is an ancestor of $TOP_{\mathcal{T}}(A)$, which implies $TOP_{\mathcal{T}}(A')$ is an ancestor of $TOP_{\mathcal{T}}(B)$. Suppose the path between A' and A is $X_0, X_1, X_2, \dots, X_k$ where $A' = X_0$ and $A = X_k$; we will show $TOP_{\mathcal{T}}(A')$ is an ancestor of $TOP_{\mathcal{T}}(A)$ by showing $TOP_{\mathcal{T}}(A')$ is an ancestor of all X_i for $i \geq 1$. Proof by induction on i . For $i = 1$, A' and X_1 share an edge, implying they appear in $\chi(t)$ together for some tree node t . By definition, $TOP_{\mathcal{T}}(A')$ and $TOP_{\mathcal{T}}(X_1)$ are both ancestors of t . Since \mathcal{T} is TOP-semiunique (so $TOP_{\mathcal{T}}(A') \neq TOP_{\mathcal{T}}(X_1)$) and \mathcal{T} is compatible with β (so $TOP_{\mathcal{T}}(X_1)$ cannot be an ancestor of $TOP_{\mathcal{T}}(A')$), this means that $TOP_{\mathcal{T}}(A')$ is an ancestor of $TOP_{\mathcal{T}}(X_1)$. For $i > 1$, we know that X_{i-1} and X_i share an edges, implying they appear together in $\chi(t)$ for some tree node t . $TOP_{\mathcal{T}}(X_i)$ and $TOP_{\mathcal{T}}(X_{i-1})$ must both ancestors t . Note that the inductive hypothesis gives us that $TOP_{\mathcal{T}}(A')$ is an ancestor of $TOP_{\mathcal{T}}(X_{i-1})$, implying it is an ancestor of t . By the same logic as before, this implies that $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(X_i)$. We thus have that $TOP_{\mathcal{T}}(A')$ is an ancestor of $TOP_{\mathcal{T}}(A)$.

We now suppose, without loss of generality, that A and B have different operators. Since $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, we know A comes before B in the compatible ordering β . However, the fact that $B \in \beta_{C+}^O$ implies that every path between B and A includes an attribute X that is either an output attribute or comes before B in β . Either way, none of these X is in the subtree rooted at $TOP_{\mathcal{T}}(A)$, implying that A and B are disconnected in the subtree rooted at $TOP_{\mathcal{T}}(A)$. This contradicts the subtree-connected property. \blacktriangleleft

► **Definition 88.** Given an AJAR problem $Q_{\mathcal{H}, \alpha}$, suppose C_1, \dots, C_k are the connected components of $\mathcal{H} \setminus (\mathcal{V}_{-\alpha} \cup \text{PA}(\alpha))$. Define a function $H(\mathcal{H}, \alpha)$ that maps AJAR queries to a set of hypergraphs as follows:

- $C_i^{++} = \bigcup_{E \in \mathcal{E}_C} E$ for all $1 \leq i \leq k$
- $\mathcal{H}_0 = (\mathcal{V}_{-\alpha}, \{F \in \mathcal{E} \mid F \subseteq \mathcal{V}_{-\alpha}\} \cup \{\mathcal{V}_{-\alpha} \cap C_i^{++} \mid 1 \leq i \leq k\})$
- $\mathcal{H}_i^+ = (C_i^{++}, \mathcal{E}_C \cup \{\mathcal{V}_{-\alpha} \cap C_i^+\})$
- $H(\mathcal{H}, \alpha) = \{\mathcal{H}_0\} \cup \bigcup_{1 \leq i \leq k} H(\mathcal{H}_i^+, \alpha_{C_i^+ \setminus \alpha_{C_i^+}^O})$

The hypergraphs in the set $H(\mathcal{H}, \alpha)$ are defined to be the *characteristic hypergraphs*.

► **Theorem 89 (Copy of Theorem 48).** *For an AJAR query $Q_{\mathcal{H}, \alpha}$ involving product aggregates, suppose $\mathcal{H}_0, \dots, \mathcal{H}_k$ are the characteristic hypergraphs $H(\mathcal{H}, \alpha)$. Then AGHDs G_0, G_1, \dots, G_k of $\mathcal{H}_0, \dots, \mathcal{H}_k$ can be connected to form a decomposable AGHD G for $Q_{\mathcal{H}, \alpha}$. Conversely, any decomposable AGHD G of $Q_{\mathcal{H}, \alpha}$ can be partitioned into AGHDs G_0, G_1, \dots, G_k of the characteristic hypergraphs $\mathcal{H}_0, \dots, \mathcal{H}_k$. Moreover, in both of these cases, $\gamma\text{-width}(G) = \max_i \gamma\text{-width}(G_i)$.*

Proof. The proof is the exact same as the proof of Theorem 31 provided in Appendix D. \blacktriangleleft

This lets us apply all the optimizations from Section 5.2, 5.3, and 5.4 to AJAR queries with product aggregates.

Comparison to FAQ

The runtime of InsideOut on a query involving idempotent product aggregations is given by $\tilde{O}(\text{IN}^{faqw})$, where the faqw depends on the ordering, and the presence of product aggregations. Our algorithm for handling product aggregations recovers the runtime of FAQ. Formally,

► **Theorem 90.** *For any AJAR query involving idempotent product aggregations, $\text{IN}^{w^*} + \text{OUT} \leq 2 \cdot \text{IN}^{faqw}$.*

The proof is in Appendix B.1. By applying ideas from the FAQ paper to our setting, we can also recover the FAQ runtime on $\#QCQ$ (Appendix E.3). Our algorithm for detecting when two orderings involving product aggregates are equivalent (Algorithm 7) is both sound and complete; in contrast, FAQ’s equivalence testing algorithm is sound but not complete. Moreover, we have a width-preserving decomposition for queries with product aggregates. This allows us to apply all the optimizations from Section 5, giving us tighter runtimes in terms of submodular and DBP-widths (Theorems 38, 39) and efficient MapReduce Algorithms (Theorems 40, 41). As shown before, FAQ gives a worse runtime exponent in each of these cases.

E.3 Recovering $\#QCQ$

We discussed idempotent product aggregations and how they can help AJAR generalize QCQ in Section 6. There is a variant called $\#QCQ$ in which solutions are expected to output the number of solutions to a given QCQ (instead of the solutions themselves). At first this seems like a fairly straightforward extension to QCQ . If we use AJAR to solve a given QCQ , the output is a relation that lists the satisfying assignments, where each tuple’s annotation is 1; to count the number of tuples, we simply need to prefix the QCQ query with aggregations using the operator $+$.

An issue arises because these new aggregations need to occur in the domain \mathbf{Z}_+ (the non-negative integers) instead of $\{0, 1\}$. Though $(\mathbf{Z}_+, \max, \cdot)$ is still a semi-ring, the product aggregations are no longer idempotent in the given domain; we discuss how to handle non-idempotent aggregation in Appendix E.4, but the added complexity (and runtime) required to deal with non-idempotent aggregations seems unnecessary in our case. Even though multiplication is not idempotent over the larger domain, we can guarantee that it is idempotent whenever a product aggregation occurs; the annotations do not leave the $\{0, 1\}$ domain until the $+$ aggregations, which must occur after the product aggregations.

To handle this extra structure, we introduce the concept of specifying restricted domains in AJAR queries. To recover $\#QCQ$, we translate the approach of FAQ [15, Section 9.5], which is the minimal application of the restricted domain concept to AJAR queries.

► **Definition 91.** Given a domain \mathbb{K} and operator set O , we define a *restriction* to subsets of the domain $\mathbb{K}_r \subset \mathbb{K}$ and operator set $O_r \subseteq O$ such that $\{0, 1\} \subseteq \mathbb{K}_r, \otimes \in O_r$ and for any $a, b \in \mathbb{K}_r$ and $\odot \in O_r$, $a \odot b \in \mathbb{K}_r$.

► **Example 92.** In the context of $\#QCQ$, $\mathbb{K} = \mathbf{Z}_+$ and $O = \{+, \max, \otimes\}$. The restriction is $\mathbb{K}_r = \{0, 1\}$ and $O_r = \{\max, \otimes\}$.

Note that if we ensure that the specified operators are closed in the restricted domain, the semiring properties will all hold in the restricted domain. We then define an aggregation ordering that incorporates these restrictions - we will define an index l divides the unrestricted and restricted portions of the ordering.

► **Definition 93.** Given an attribute set \mathcal{V} , domain \mathbb{K} , operator set O , and restriction \mathbb{K}_r and O_r , an *restriction-compatible* aggregation ordering is an aggregation ordering α and index l such that $1 \leq l \leq |\alpha|$ and for each $k \geq l$, $\alpha_k = (A, \odot)$ for $A \in \mathcal{V}$ and $\odot \in O_r$.

Any single operator \odot that appears both before and after the division index l will be treated as different operators (this issue does not come up in the context of $\#QCQ$). We can then define an AJAR query to use a restriction-compatible ordering, and any instance of the query must have \mathbb{K}_r -relations. Under this definition, we can treat the product aggregations as idempotent, allowing us to use the work in Section 6 to recover $\#QCQ$.

This set-up is essentially a translation of FAQ’s results to our language/notation. Using the exact same construction described in the previous Appendix section, we can now recover FAQ’s runtime on $\#QCQ$ as well. We note that we could extend this idea of restricting domains even further by relying on our GHDs. In particular, we can have every single element of the aggregation ordering specify its own domain, and a valid GHD would have to ensure that for any A, B such that $TOP_{\mathcal{T}}(A)$ is an ancestor of $TOP_{\mathcal{T}}(B)$, the semiring domain corresponding to A is a superset of the semiring domain corresponding to B .

E.4 Non-Idempotent Product Aggregations

Our AggroYannakakis algorithm actually implicitly assumes that any product aggregation that arises consists of an *idempotent* operator.

► **Definition 94.** Given a set S , an operator \oplus is *idempotent* if and only if for any element $a \in S$, $a \oplus a = a$.

This is a reasonable assumption, as the problems that we’ve discovered using product aggregation all tend to have idempotent products. The key difference between an idempotent and non-idempotent operator is the distributive property; $(a \otimes b) \otimes (a \otimes c) = a \otimes (b \otimes c)$ only if \otimes is idempotent. Note that the non-idempotent case would require an a^2 . So, to be complete, we can support non-idempotent operators by raising the annotations of every other relation to a power. In particular, if we have a non-idempotent aggregator for an attribute A , we should raise the annotations for the relations in every other node in our tree to the $|\mathcal{D}^A|$ power when we aggregate the attribute A away.

F Extension: Computing Transitive Closure

A standard extension to the basic relational algebra is the transitive closure or Kleene star operator. In this section, we explore how our framework for solving AJAR queries can be applied to computing transitive closures. First we define the operator using the language of AJAR. Given a relation R with two attributes, consider the query

$$Q_k = \sum_{A_2} \cdots \sum_{A_k} \bowtie_{1 \leq i \leq k} R(A_i, A_{i+1})$$

where each of $R(A_i, A_{i+1})$ are identical copies of R with the attributes named as specified. Note that our output Q_k is going to be a two-attribute relation. Suppose there exists some k^* such that Q_k is identical for all $k \geq k^*$. We can then define the transitive closure of a relation R , denoted R^* , to be Q_{k^*} .

This classic operator has natural applications in the context of graphs. If our relation R is a list of (directed) edges (without meaningful annotations), computing R^* is equivalent to computing the connected components of our graph. If we add annotations over the semiring $(\mathbb{Z} \cup \{\infty\}, \min, +)$ where each edge is annotated with a weight, then computing R^* is equivalent to computing all pairs shortest paths [8]. Note that we can guarantee R^* exists as long (i) our graph contains no negative weight cycles and (ii) our relation contains self-edges with weight 0. We will discuss computing R^* in the context of graphs, applying it to the all pairs shortest path problem. Let E be the number of edges and V the number of nodes in the graph; we will derive the complexity of computing all pairs shortest paths in terms of E and V .

A naive algorithm for finding R^* is to compute Q_1, Q_2, Q_4, \dots until we find two consecutive results that are identical. This approach requires answering $O(\log k^*)$ AJAR queries. In the context of all pairs shortest path, we know $k^* \leq V$, which means that the number of queries to answer is $O(\log V)$. We start by analyzing the computation required to answer a query of the form Q_{2^n} .

We define the GHD to use for Q_{2^n} recursively. Our base case, when $n = 1$, is to have a single bag containing all three attributes A_1, A_2, A_3 . For $n > 1$, the root of our GHD will contain the attributes $A_1, A_{2^{n-1}+1}, A_{2^n+1}$. It will have two children: on the left, it will have the GHD corresponding to $Q_{R^{2^{n-1}}}$, and on the right it will have an identical GHD over the attributes $A_{2^{n-1}+1}, A_{2^{n-1}+2}, \dots, A_{2^n+1}$ instead of the attributes $A_1, A_2, \dots, A_{2^{n-1}+1}$. Note that each bag of our constructed GHD has 3 attributes, but they may not appear in any relation together. Additionally, note that the depth of our GHD is simply n .

If we naively apply the AGM bound to derive the fractional hypertree width, we get a width of E^3 . However, if, for each attribute A_i , we (virtually) create a relation $S(A_i)$ of size V , our fractional hypertree width becomes V^3 . Alternatively, we can also use DBP-width to derive the V^3 bound without introducing these relations.

Applying the results of GYM [3] gives us that we can answer Q_{2^n} in $O(n)$ MapReduce rounds with $O(V^3)$ communication cost. Given that we need to answer $O(\log V)$ of these queries and that $n \leq O(\log V)$ for each of these queries, we have a $O(\log^2 V)$ round MapReduce algorithm with $\tilde{O}(V^3)$ total communication cost for all pairs shortest paths, which is within poly-log factors of standard algorithms for this problem.

In addition, if we allow a $O(k^* \log k^*)$ round MapReduce algorithm, we can reduce the total communication cost to $\tilde{O}(EV)$ by using a chain GHD. In particular, for a query Q_k , the GHD will be a chain of k bags such that the i^{th} bag in our chain consists of A_i, A_{i+1} and A_{k+1} . This construction ensures that two of the three attributes in each bag appear in a relation together, reducing the width to EV .

We note that we derived this MapReduce bound with our generic algorithms, without any specialization for this particular problem. We can also derive a serial algorithm for the problem with the same bound, but it requires a small optimization. By construction, our (original, non-chain) GHD has the property that every subtree whose root is at a particular level is completely identical. This means that AggroGHDJoin does not need to visit each bag; it simply needs to visit one bag per level, and then assign the result to the other bags on the level. With this optimization, our algorithm computes all pairs shortest paths in $\tilde{O}(V^3)$, again within poly-log factors of specialized graph algorithms.